Attentiveness: Reactivity at Scale

Gregory S. Hartman

CMU-ISR-10-111
December 2010

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

William Scherlis, Chair
Len Bass
David A. Eckhardt
Bruce Horn, Microsoft

This material is based upon work supported by: the Software Engineering Institute; Yahoo!; Lockheed Martin: RRMHS1798; ARO: DAAD190210389; CyLab: W911NF0910273.

Keywords: Reactive systems, responsiveness, state consistency, concurrency, distributed systems, data race detection, cancel, rollback

Abstract

Clients of reactive systems often change their priorities. For example, a human user of an email viewer may attempt to display a message while a large attachment is downloading. To the user, an email viewer that delayed display of the message would exhibit a failure similar to priority inversion in real-time systems.

We propose a new quality attribute, **attentiveness**, that provides a unified way to model the forms of redirection offered by application-level reactive systems to accommodate the changing priorities of their clients, which may be either humans or system components. Modeling attentiveness as a quality attribute provides systems designers with a single conceptual framework for policy and architectural decisions to address trade-offs among criteria such as responsiveness, overall performance, behavioral predictability, and state consistency.

At the policy level, the framework models diverse redirection options including cancel, undo, defer, checkpoint, and ignore. At the architectural level, the framework includes concepts such as: distinguishing "short" operations (e.g., an event notification) from "long" operations (e.g., unbounded data transfer over a network); encapsulating long operations to prevent interference with redirection; enabling use of light-weight checkpoints to support redirection while executing "long" operations; and consolidating responsibility for redirection to a small group of components in the system. Policy and architecture come together in the form of a set of positive and negative patterns for realizing attentive systems. These patterns are derived from case studies of attentiveness failures and successes, several of which are presented and evaluated in this paper.

The value of the framework has been tested through experiments involving both new development and re-engineering existing projects. We present two of these experiments in this paper, including both human-system interaction in a document editor and system-system interaction in a client-server application. These experiments illustrate that our modeling framework can guide incremental attentiveness improvements in existing reactive systems.

Acknowledgments

I would like to express my appreciation for the help and encouragement that I received from:

William Scherlis
Len Bass
David A. Eckhardt
Bruce Horn
Jonathan Aldrich
Tamara and Greg Beckenbaugh
Nels Beckman
Emanuel Bowes
Brian S. and Michelle M. Butler
Robin Capcara
Catherine Copetas
Genevive Craver
Tom Craver and Jan Rose
Chris Dalansky
Fr. Michael Darcy
Monika DeReno
John M. Dolan
David A. Dorsey
Nancy Drew
George Fairbanks
Tracy Fasick
Nick Frollini
Lois Fyke
David Garlan
Robert B. Griffiths
Thomas and Theresa Grosh
David and Libby Hall
Tim and Ellen Halloran
Christian Hallstein
David Handron
Charles and Betty Hartman
Gary and Sherry Hartman
Rebecca Hartman
Richard and Susan Hartman
Tom and Melody Hartman
Jordan Hayes
Connie Herold
Helen Higgins
Tom Hillman
Joe and Nancy Huerter
Ciera and Saul Jaspan
Bonnie E. John
Jeanne and Andy Kohn
Tina Lockett
Barry Luokkala
Katherine A. Lynch
Larry Maccherone
Eric Mann
Paul Mazaitis
Donald and Joan McBurney
Christoph Mertz
Mabel Millen
Arlene B. Miller
Ryan Murphy
Ann Myers
Priya Narasimhan
Luis Navarro-Serment
Gary D. Patterson
Holly Puett
Jack and Lucy Scandrett
Mary Shaw
Dean F. and Elizabeth Sutherland
Dwight and Carol Thomas
Peter and Kelly Venable
Cliff and Mary Beth Wagner
Jack Walsh
Andrew Wesie
Jeannette M. Wing

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Software systems occasionally fail to respond to their "clients," either human users or other systems. For example, a user attempting to place a call with a touch screen cellular phone could encounter several problems. The phone could:

- Ignore some of the user's interactions with the touch screen
- Display buttons that appear to be held after the user has released them
- Display a black screen for many seconds, refusing to accept input
- Display information irrelevant to the user's activity, such as a new e-mail notification

All of the problems mentioned above can be observed when placing a call with a HTC Droid Eris running Android 2.1 software, and delay the process of making a call for one minute. In addition, these problems have secondary effects, including increased error rates [10], increased state anxiety [48], and decreased satisfaction with the overall system [92]. Similar failures can be observed in systems as diverse as hand-held devices, desktop applications, and servers.

In this work we propose a quality attribute called *attentiveness* that describes the ability of some systems to avoid these problems by changing their computational trajectory in response to requests from their clients. Attentiveness describes the relationship between requests and the system's responses in terms of promptness and consistency. On a phone, users create requests by touching the phone's screen. The phone responds both by updating the screen and also by connecting and disconnecting calls. Promptness assesses the delay between the touch and the phone's response. Consistency assesses both the predictability of the phone's responses and also the phone's ability to preserve the user's prior work. Therefore, the phone exhibits a consistency failure if it changes a button just as the user is touching it. For example, on the phone described above, the "End Call" button becomes the "Call" button when a call ends, as shown in Figure 1.1. If the user were to press this button just as the other party hangs up, he would accidentally place a new call. In addition, the phone would exhibit a consistency failure if it were to crash while the user was dialing a number. To avoid crashing the phone must maintain harmony among elements of its internal state while responding to new requests.

Attentiveness failures are not confined to systems that interact directly with users. Attentiveness failures occur on servers when clients disconnect after submitting long running requests. For example, when a user initiates a search in Thunderbird [99], a popular email client, Thunderbird forwards the search to the IMAP server that holds the user's messages. While the search is happening the user may decide to change the search terms. The IMAP protocol does not support interruption of search requests. Therefore, Thunderbird is forced to submit a second search request to the IMAP server. The second search will compete with the initial, now unwanted, search for resources on the IMAP server, as shown in Figure 1.2. As a result, the

Figure 1.1: On some cellular phones the "End Call" button transforms into a "Call" button when the other party hangs up. A user attempting to use the button may reestablish a call with the other party by mistake.

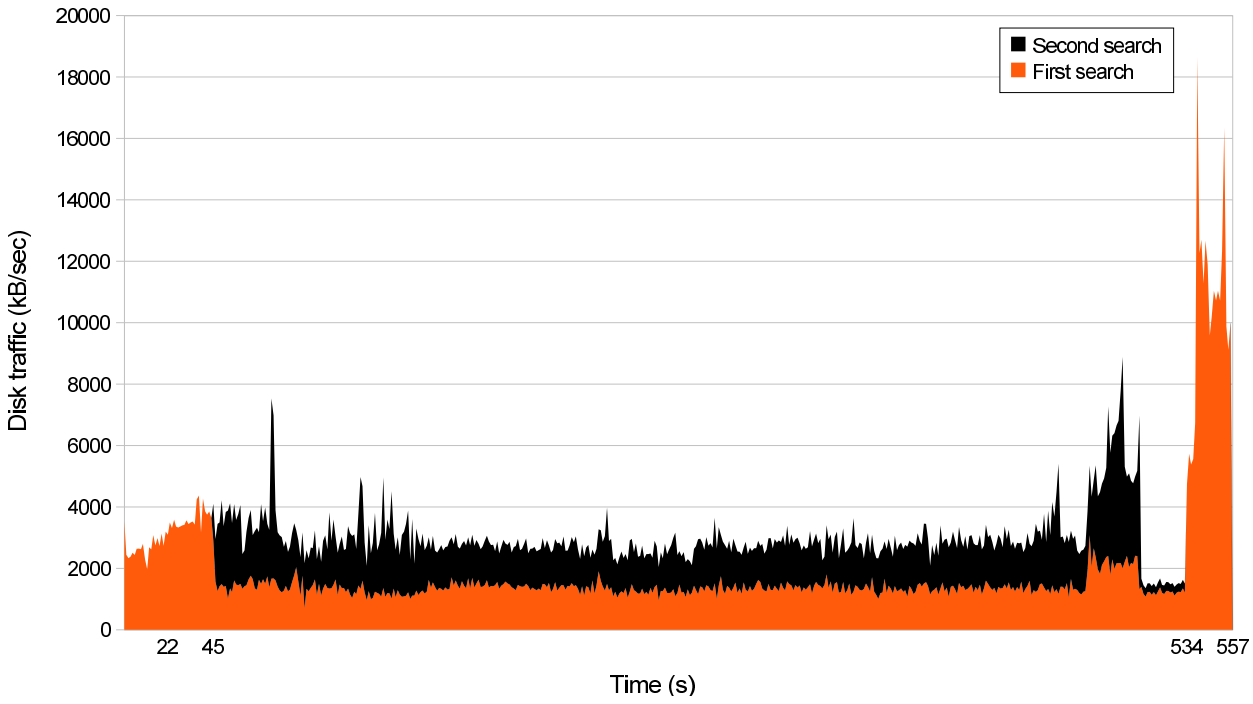


Figure 1.2: The IMAP protocol provides no way to cancel search requests. Therefore, if a user changes search terms while a search is in progress the e-mail client will send a second search to the IMAP server. The IMAP server divides its disk bandwidth between the searches. As a result, both searches are delayed.

user's search is delayed.

Attentiveness draws on insights from four areas of research:

- Human-computer interaction contributes knowledge of the relationship between response times and human satisfaction with systems. Attentiveness is most closely related to direct manipulation [94].
- Software architecture both contributes architectural patterns such as model-view-controller [63] and also has identified the phenomenon of architectural mismatch [39] that makes attentive systems difficult to construct.
- Systems software contributes knowledge of the problems of concurrency, concepts for reasoning about concurrency such as the happens-before relationship [66], and approaches to maintaining consistency in the presence of concurrency such as transactions [42, 56].
- Software monitoring, especially prior work in checking for data races [86], provides techniques for observing software as it executes, and also provides techniques to control the system's execution to preserve consistency [83].

In this work we present: a framework for creating requirements for attentiveness, directives that describe attentiveness in terms of implementation, a design pattern for attentive systems, and the design of two runtime systems to support attentiveness.

1.1 Conceptual contributions

We propose the following concepts to address attentiveness in a systematic and general way:

- **Requests** relate the activity of a system to inputs from its client. In threaded systems each thread is associated with a request. Designers are able to identify **conflicting** requests that must not execute concurrently due to the system's contract with its users. For example, Thunderbird must not attempt to display the contents of two messages in the same message pane.
- **Behaviors**—such as CANCEL, PAUSE, and INTERRUPT—define general approaches that clients can use to redirect systems. For example, a user of an email client may PAUSE a download to free network bandwidth in order to display a high priority message.
- **A calculus of short and long operations** allows developers to reason about promptness qualitatively.
- **Directives** allow architects and developers to represent knowledge that cannot be inferred from the system's implementation. For example, directives identify **regions**—parts of the system's state that must be treated as a unit from the perspective of consistency. Developers and runtime systems can use this knowledge to improve the attentiveness of systems.
- **Trusted execution** uses knowledge provided by directives to provide services that redirect threads. Unlike existing approaches to redirecting threads, the services are able to redirect threads promptly while preserving the system's consistency. Developers can utilize these services to simplify the implementation of the behaviors mentioned above. In addition, trusted execution identifies **dependencies** among requests. Dependencies are created when requests access regions. The runtime system tracks dependencies to avoid introducing check-then-act errors when it redirects threads. It accomplishes this by using the dependencies to identify the group of threads that have observed the activity of the thread being redirected, and then redirecting every thread in the group. Developers can reduce the size of these groups by introducing additional directives in the system, allowing them to trade off development effort for greater efficiency.
- **Architectural structures**, shown in Figure 1.3, track requests in the system, allow developers to manage the assignment of resources to requests, and confine the responsibility for implementing be-

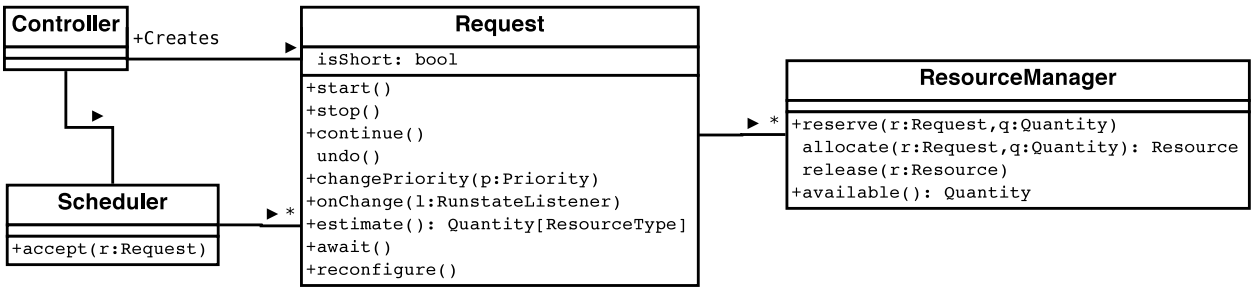


Figure 1.3: Architectural structures for attentiveness

```
1  writer opaque FILE * fopen(borrowed_ro const char * filename,
2                             borrowed_ro const char * mode);

4  reader int feof(opaque FILE * fp);

6  independent transparent void * memcpy(borrowed_rw void * dstpp,
7                                        borrowed_ro const void * srcpp,
8                                        size_t len);
```

Listing 1.1: Partial tollgate for the C runtime library. The tollgate is defined by the modifiers shown in bold type.

haviors to a small number of components.

- **Checked execution** monitors the system as it executes, ensuring that its behavior is consistent with the information provided by directives. During checked execution, threads gain and release **permissions** to read and write regions. Checked execution replaces the threads in the system with **filaments.** Like threads, all of the filaments in a system share a common address space. Unlike threads, filaments have unique sets of permissions that are granted and revoked by directives.
- **Attentive protocols** allow clients to redirect requests submitted to servers and allow servers to detect client failures promptly.
- **Tollgates** allow developers to attach additional information at the interface between the system and third-party modules that do not contain directives, as shown in Listing 1.1. Trusted execution uses the information provided by tollgates to manage dependencies among requests that access the modules. Checked execution verifies that the information provided by tollgates is accurate.

1.2 Impact on practice

The key concepts of attentiveness affect four stages of software development:

In **interface design**, which includes both human-system interfaces and system-system interfaces, behaviors provide standard patterns for redirecting requests. Behaviors point to tradeoffs that designers should consider when creating requirements for the system. In addition, they allow designers to provide concise, unambiguous requirements to developers.

Software architectures that incorporate our design pattern both provide a central location for tracking requests and also encapsulate responsibility for implementing behaviors to a small collection of components. As a result, architects do not need to consider attentiveness when designing other parts of the system. In addition, directives, described in Chapter 3, allow architects to specify constraints on the implementation of

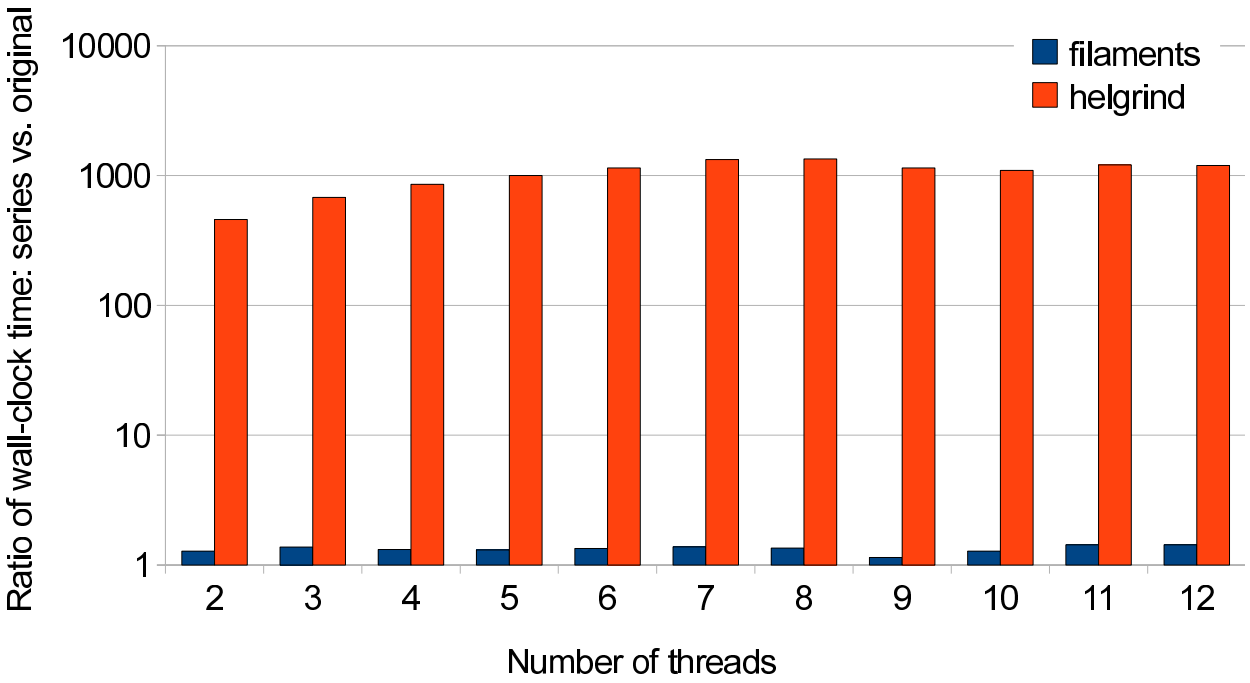


Figure 1.4: Directives improve the efficiency of software monitoring. This graph compares checked execution to Helgrind [89]. Both systems are dynamic checkers that detect data races. Checked execution is faster by a factor of 300.

the system that can be enforced through checked execution.

Developers benefit from the knowledge conveyed by the directives specified by architects. In addition, the encapsulation of attentiveness concerns into a small number of architectural elements allows developers to ignore attentiveness when implementing most of the functionality in their systems. Finally, the runtime system greatly simplifies the task of redirecting requests. In current practice, developers are forced to redirect requests using either operations that interrupt threads promptly without preserving consistency or operations that preserve consistency but may take an unbounded amount of time to interrupt a thread. The runtime support described in Chapter 5 provides operations that are both prompt and also preserve consistency. Micro-benchmarks, also described in Chapter 5, indicate that redirection can be prompt given the definition of promptness used by many interactive systems.

Finally, **software monitoring** allows developers to quickly identify inaccurate directives and also could allow detailed failure reports to be submitted from deployed systems. Without software monitoring, developers would often have to work backward from a system failure to identify one or more inaccurate directives, a process that is often both tedious and error prone. Software monitoring also benefits from the presence of directives. Since the directives predict the system's future behavior, the software monitor is able to be highly efficient when compared to software monitors that do not rely on directives, as shown in Figure 1.4. In addition, the presence of directives allows the software monitor to avoid generating false positives.

1.3 Scientific conclusions

The principal hypothesis of this work is that, from the standpoint of both requirements and architecture, *attentiveness can be addressed in a systematic fashion.* We first propose a design pattern that addresses

properties of requests that are directly related to attentiveness. Next, we define directives that describe important properties of the design pattern that must be preserved by implementations. Finally, we define runtime support that assists developers when developing attentive systems. We evaluate four aspects of this hypothesis in this work:

Chapter 3 evaluates the directives to determine if they can describe the regions of third-party systems by applying directives to six benchmarks taken from the PARSEC 2.0 benchmark suite [13]. One of the benchmarks employs data races as part of its design and cannot be modeled with directives. The other five benchmarks can be modeled with directives. However, limitations in our current implementation of checked execution make it impossible to verify the accuracy of the directives in two of the benchmarks.

In Chapter 4 we evaluate our design pattern to determine if it resolves attentiveness failures. We apply the design pattern to a Java client and server connected with the Remote Method Invocation [96] communication protocol. The design pattern resolves a reproducible attentiveness failure in this system with a runtime overhead of 5%.

In Chapter 5 we test the runtime system to verify that checked execution can be implemented efficiently. We use the PARSEC benchmarks described above in this test, observing the memory consumption and execution time of the benchmarks. In the worst case the execution time of the benchmarks increases by a factor of 3 under checked execution, and the memory overhead increases by a factor of 2. As a result, it is possible to use additional cores to overcome the overhead of checked execution.

In Chapter 5 we describe micro-benchmarks to verify that the overhead of trusted execution would be acceptable. These micro-benchmarks indicate that the operations needed to maintain consistency can be long for large regions, but complete in approximately one second in the worst case. Systems using our proposed design pattern will remain attentive in the presence of these operations.

1.4 Scope

This work does not exhaustively cover all aspects of attentiveness. There are a number of opportunities for further development of the ideas outlined in this work:

The approach described in this document does not guarantee that a system will be free of attentiveness failures

There is no guarantee that systems constructed with the techniques described in this document will be attentive. These techniques allow developers to incrementally improve the attentiveness of systems, use third party code without analysis, and use non-deterministic resources such as networks and disks. Techniques used for hard real time system development, such as worst case execution time analysis, would provide stronger guarantees of the behavior of the system, but would require more up-front development effort.

System calls and attentiveness

An attentive system may need to redirect requests that are engaged in system calls. It may be possible to redirect these requests by creating a mediator to manage the effects of the calls. The design and implementation of system call mediators is not within the scope of this research.

Handling irreversible changes

It may not be possible to reverse all of the changes made by a request when it is redirected. The definition of attentiveness specifies that any remaining changes must be communicated to the client when a request is redirected. The current design does not provide a way to detect these changes and communicate them to the client.

Implementation and evaluation of the overhead of trusted execution

This work provides a detailed design of trusted execution, including the algorithms that are needed to manage dependencies. However, we have not implemented this system and have not evaluated the overhead imposed by our techniques.

Improving the efficiency of checked execution

There are several techniques that could improve the efficiency of checked execution both in terms of memory consumption and also in terms of execution time.

Evaluation of behaviors in the context of interface design

We have not conducted studies to ensure that designers can employ our behaviors when designing systems or to assess the effect of behaviors on the quality and complexity of system designs.

Addressing attentiveness in systems that are not request-oriented

Some classes of systems, including simulators, respond to input from their clients but are not easily modeled in terms of discrete requests. We have not attempted to assess these systems in terms of attentiveness.

Increasing certainty that we have the right set of directives

We have not attempted to verify that our list of directives is complete. There are some indications that new types of directives may be useful. For example, it may be helpful to use directives to track the right to deallocate memory in systems that use explicit memory management. In addition, it may be helpful to provide directives to document aliasing assumptions in systems.

Reducing developer effort when introducing directives into systems

Introducing directives into systems requires considerable developer effort. There are several approaches that could reduce the effort of introducing directives, including static analysis. It would also be useful to have an analysis that would identify contradictory directives.

Advice for constructing new protocols that support attentiveness

Inter-application communication protocols can support attentiveness by explicitly supporting the behaviors described in this research. However, it may be possible to further improve the attentiveness of protocols by changing the way that state changes are communicated between clients and servers. Section 4.1.4 identifies

several problems in the IMAP protocol related to state management. The description of a solution to the problems of state management is beyond the scope of this research.

Reverse tollgates

Tollgates assume that the caller will have directives and the code being called will not have directives. However, in systems that exhibit inversion of control, the code without directives may act as the caller, creating a reverse tollgate. The directives described in this document are likely to be applicable to reverse tollgates. However, the implementation of the tollgate is likely to be somewhat different.

1.5 Roadmap

A detailed discussion of attentiveness is contained in Chapter 2 through Chapter 5. The discussion starts at the level of requirements in Chapter 2, gradually moving to the level of detailed implementation in Chapter 5.

Chapter 2 discusses the concept of attentiveness at the level of requirements. It begins by providing a detailed description of promptness and consistency. Next, it describes redirection in terms of attributes of requests, including conflicting requests and priorities. Then it describes the concept of dependencies. Finally, it describes a series of challenging problems that must be addressed by attentive systems. The key concepts covered in Chapter 2 include: requests, behaviors, conflicting requests, priorities, and the calculus of short and long operations.

Chapter 3 describes the concept of directives. First, it describes directives relating to consistency, including the directives that define regions and control a thread's permissions to access regions. Next it describes directives related to promptness, including directives that allow developers to identify short blocks of code that will execute to completion in the event of redirection. Then it discusses directives that identify requests, control dependencies among requests, and associate requests with threads. Finally, the chapter describes tollgates and their relationship to module interfaces. The key concepts covered in Chapter 3 include: directives, regions, permissions, dependencies, and tollgates.

Chapter 4 describes the relationship between attentiveness and the design of systems. It starts by assessing the designs of six third party systems in terms of attentiveness, relating the designs to observations of the runtime behavior of the systems. Next the chapter proposes architectural components that can be added to systems to improve their attentiveness. It concludes by applying these architectural components to a small system, demonstrating that they lead to an improvement in attentiveness. The key concepts covered in Chapter 4 are the architectural structures for attentiveness and the concept of attentive protocols.

Chapter 5 describes the design and implementation of runtime systems to support attentiveness. First the chapter describes the design of a runtime system to support trusted execution. Next the chapter describes the design and implementation of a runtime system that supports checked execution, verifying the accuracy of the directives related to consistency. The key concepts covered in Chapter 5 include: trusted execution, checked execution, filaments, and tollgates.

Chapter 2

Attentiveness

Many systems are not responsive to their clients, delaying their work. For example, a user of Thunderbird 2.0, an email program, may want to find the location of a meeting that will begin at 10:00AM. Therefore, at 9:55AM the user starts Thunderbird, identifies the email containing the invitation, and asks Thunderbird to display the message. However Thunderbird notices a large number of new messages in the user's Inbox. It automatically begins to download the new messages to scan them for junk mail without a request from the user. Normally, this behavior is beneficial: Thunderbird is able to remove junk messages so that they do not distract the user. However, in this situation Thunderbird's scanning of junk messages delays the message view requests for more than 5 minutes, causing the user to be late for the meeting. The cost of these delays to human users can be amplified by increased error rates by human operators [10] and increased state anxiety [48].

The delay introduced in this scenario is similar to the priority inversions that can be encountered in defective real-time systems. The user's request to display the invitation was a high-priority task and should have preempted the junk mail scanner. In this chapter we propose a new quality attribute called attentiveness that describes functionality that can resolve this problem. An attentive system would have addressed this scenario by either:

- Automatically prioritizing requests. For example, the system could identify every *ViewMessage* request as a high priority task, giving *ViewMessage* requests precedence over the junk mail scanner
- Allowing the user to prioritize the task by both providing an overview of work in progress and also allowing the user to redirect the system

Similar functionality could improve a wide variety of systems, including both interactive applications, where the clients are humans, and also servers, where the clients are other systems. Some systems both send and receive requests. For example, an email program like Thunderbird receives requests from its users and also sends requests to the IMAP server that holds the user's messages. In this situation the design of the IMAP protocol may limit the email program's attentiveness as discussed in Chapter 4. For clarity, the majority of the examples in this chapter will discuss an email program interacting with a human user and an IMAP server.

In this chapter we discuss attentiveness incrementally, starting with aspects of attentiveness that are directly observable and gradually moving to constraints on the system's implementation. In Section 2.1 we propose an abstract model of the communication between clients and systems. The model represents communication in terms of discrete requests generated by the client and the system's responses to these requests. Clients do not have to wait for the system to complete prior requests before submitting new ones. The system's behavior is predictable when clients and systems agree on two properties of requests developed in this section: conflicts and priorities.

The model of requests, responses, conflicts, and priorities draws extensively on several areas of prior work. In Section 2.2 we discuss four areas that have influenced the model, both describing ideas that we have adapted and also contrasting attentiveness with problems addressed in this work. The areas include:

- User interface design
- Computer supported collaborative work (CSCW)
- Transactions, including databases and transactional memory
- Real-time system design

In Section 2.3 we discuss the aspects of the system's communication that are directly related to time. **Promptness** refers to a system's ability to respond to requests within a period of time that is acceptable to its clients. Systems may both send multiple responses for a request and also process multiple requests in parallel. Therefore, we define multiple measurements to assess the promptness of individual requests.

In Section 2.4 we address promptness in the context of sequences of requests. Systems can improve their promptness by redirecting some of the requests in the sequence. We define general patterns of redirection, called **behaviors**, that apply to many systems. Systems employ behaviors when it is not possible to run all of the requests submitted by clients in parallel and the system must redirect work in progress to admit a high priority request. System designers and clients choose the behaviors to invoke, while the system chooses the requests to redirect based on conflicts and priorities.

In Section 2.5 we consider problems that clients may encounter when redirecting systems. Clients should be able to redirect systems without either losing access to the system or losing work. We describe a property called **consistency** that describes two aspects of systems from a client's point of view. In addition, consistency has implications for the implementation of the system. Externally, the system must continue to provide predictable responses to requests submitted by its clients. In addition, the system must preserve completed work, as much of the work in progress as possible, and the future work done by the client. To accomplish this, the system must maintain both the invariants of its data structures and also the invariants that govern its communication with other systems.

In Section 2.6 we expand the model of requests to describe the implementation of systems in general terms. The model contains four operations that control requests: start(), stop(), continue(), and undo(). These operations are sufficient to implement the behaviors mentioned in Section 2.4.

The model assumes that one or more threads in the systems process requests. It assumes that threads can observe the partially completed changes made by threads processing other requests. It models these observations as **dependencies** among requests. To avoid check-then-act failures, the model specifies that calling undo() on a request will roll back any other request that has observed its changes.

Finally, the model employs a qualitative approach to reasoning about promptness: a **calculus of short and long operations**. This calculus applies to the operations executed by threads as they execute requests. The goal of the model is to ensure that the threads responsible for making prompt responses to threads are not blocked for unbounded amounts of time.

The model discussed in this chapter forms the basis for the chapters which follow. In Chapter 3 we discuss directives, an approach to checking the conformance of a system's implementation to the model. In Chapter 4 we describe architectural elements that implement the model by providing a mapping between threads and requests, tracking resources in the system, and isolating certain threads from long operations. In Chapter 5 we describe the design and implementation of runtime support that aids developers in implementing the start(), stop(), continue(), and undo() operations.

In Section 2.7 we describe requirements for the implementation of attentive systems.

2.1 Describing requests: conflicts and priorities

Our model assumes that communication between a system and its clients is characterized by a series of discrete requests. Clients do not have to wait for the system to complete prior requests before submitting new ones. This module describes many systems, including:

- Servers, where clients can use multiple connections to send requests
- Handheld devices, where users can initiate requests by touching the screen and pressing buttons
- Interactive applications, where users can initiate requests by interacting with controls on the screen

Some systems do not follow this model. For example, simulators and embedded sensors are often implemented with cyclic executives [8], where the system gathers input and processes the input at pre-defined intervals.

The requests in our model have two attributes that are relevant to the communication between the system and its clients: conflicts and priorities. Conflicts define pairs of requests that would not produce predictable results if they were executed concurrently. As a result, systems are forced to choose one of the conflicting requests to execute first. Priorities govern the system's choice of requests. Assuming that resources are available, the system will always execute the request with the highest priority. If resources are available, it may also execute additional requests with lower priorities that do not conflict with the highest priority request.

2.1.1 Conflicts

Conflicts specify that certain combinations of requests cannot execute simultaneously because doing so would make their effects, as defined by the system's interface, unpredictable. For example, in an email client a request to move all of the messages from $Folder_A$ to $Folder_B$ would conflict with a second request to move messages from $Folder_C$ to $Folder_A$.

If the system were to execute these requests simultaneously, the system could divide the messages initially contained in $Folder_C$ between $Folder_A$ and $Folder_B$ depending on the relative progress of the requests. These would not be true for two requests where the destination folders do not overlap the source folders. For example, a move of messages from $Folder_A$ to $Folder_B$ can safely execute in parallel with a move of messages from $Folder_C$ to $Folder_D$.

Systems cannot automatically detect conflicting requests. Therefore, designers must specify which requests, if any, in the system's interface conflict. As in this example, conflicts among requests may be conditional, depending on the parameters of the requests.

2.1.2 Priorities

Designers attach priorities to requests to control the system's behavior when it cannot execute every request submitted by its clients. Designers should choose priorities to minimize the need for clients to redirect systems. For example, in an email program a user may attempt to display a message while a download of an attachment is in progress. In this situation it is most likely that the user intends for the attempt to display the message to interrupt the download of the attachment. Therefore, the designers should attach a high priority to message view requests and a lower priority to attachment downloads.

No set of priorities can be perfect. For example, the download of the attachment may be urgent. The user may be reading messages only because the attachment is not available, but may not want this activity to delay the download of the attachment. In these situations, the user will need to override the priorities specified by designers by invoking one or more of the behaviors described in Section 2.4.

2.2 Prior work related to attentiveness

Our model of requests, behaviors, and priorities draws on several areas of related work, including: user interface design, computer supported collaborative work (CSCW), transactional databases, and real-time system design. Each of these contexts contributes concepts that we employ in our approach to attentive systems. However, the problems encountered in designing attentive systems are subtly different than the problems encountered in these systems.

2.2.1 User interface design

User interface designers have defined many of the behaviors in our model and have pointed to the need for consistent and prompt redirection. For example, a user may print a document and then decide to cancel the printout. Interface designers will specify that the system should offer a cancel button both to allow the user to regain control of the system and to stop the printout promptly. They note that systems rarely implement these features well [21]. For example, many systems forwarded multiple pages to the printer before the cancel button is pressed. When the communication protocol between the system and the printer does not support redirection, pages will continue to emerge from the printer after the user has canceled the printout.

We build on the work done by interface designers, exploring the effects that these behaviors have both on the architecture of systems and also on protocol designs. Software architects have long realized that scenarios like the one described above have implications for software architecture [11]. We build on this work, considering the implications of behaviors on system implementation and protocol design. In the process of doing this work we identify a series of design decisions that should be addressed by system designers. The concepts of requests, priorities, and conflicts allow us to do this in general terms, moving beyond prior approaches that considered the design of the system's interface in system-specific terms [53].

2.2.2 Computer supported collaborative work (CSCW)

The communication patterns of an attentive system are partially asynchronous: clients are able to submit new requests while the system is busy. Some of the new requests submitted by the client may change or override prior requests. In this respect an attentive system is much like a CSCW system processing requests from multiple users. The field of computer supported collaborative work has defined formal approaches to analyzing requests, including a calculus that can be used to identify and in some cases resolve conflicts among requests [22].

The approaches developed for CSCW are closely related to the process of choosing behaviors and identifying conflicts among requests. We do not explore this process in detail as a part of this work. However, many of the approaches outlined in CSCW reason about requests in terms of priorities and conflicts. Therefore, we assume that interface designers will provide two functions as part of their specification. One function, $\theta(r)$ accepts a request and returns the request's priority. A second function, $\kappa(r_1, r_2)$ accepts two requests and returns true if the requests conflict. In the spam scanning example the following relationship would hold:

$$\theta(\mathit{ViewMessage}(m)) > \theta(\mathit{JunkMailScan}(f))$$

In the message copy example the following rule would specify that *MoveMessage* requests with overlapping folders conflict:

$$((f_1 = f_4) \vee (f_2 = f_3)) \implies \kappa(\mathit{MoveMessages}(f_1, f_2), \mathit{MoveMessages}(f_3, f_4))$$

The implementation of the system must ensure that κ is false for all possible pairings of requests in the system. In addition, the request with the highest priority, as returned by θ, must be running. The

designs and implementations that we propose in this work have these properties. We note that, while the notation given above is precise, it may not be the optimal representation to use for capturing the system's requirements and conveying them to developers.

2.2.3 Transactions

The concepts of conflicting and prioritized requests outlined above may remind the reader of transactional systems. Indeed, transactions have been used to solve similar problems in database systems [42], shared memory multi-threading [56], and distributed systems [87]. Transactions offer two concepts that are especially interesting from the perspective of attentive systems: isolation and rollback.

Many systems isolate transactions, preventing one transaction from seeing the partially completed changes of another transaction. Therefore, when viewed by its clients, the system will behave as if it had processed the requests submitted to it in some serial order. This serial order does not necessarily have to match the order in which the requests arrived [72].

Isolation has two advantages. First, it simplifies the concurrency model for clients, since they can be confident that the system's state will not change in the middle of a transaction. In addition, clients know that their transactions will either succeed or fail with no change to the system's state. As a result, clients will not encounter check-then-act failures when using transactions. The abstract model provided by transactions allows systems to exclude these errors without knowledge of the application-specific semantics of the transaction [64]. Second, isolation avoids the problem of cascading rollbacks, where a transaction that rolls back causes other transactions that have observed its changes to also roll back.

However, isolation is not free. Designers of relational databases are aware of a tradeoff between the level of isolation provided to concurrent transactions and the performance of the database [12], and have responded by implementing more modest forms of isolation that exchange some degree of consistency for improved performance [43].

Other aspects of attentive systems make isolation less desirable:

- Rollbacks are likely to be rare and would not affect a large number of requests even in the presence of a cascading rollback
- Attentive systems may rely on non-transactional subsystems that do not offer rollbacks, such as IMAP servers
- Attentive systems may need to inform clients of the progress of their requests. This feedback would need to be treated as a special case from the perspective of isolation
- Attentive systems may use multiple communication channels, some of which may not honor isolation

The presence of multiple communication channels leads to a phenomenon called architectural mismatch. In one case architectural mismatch occurred when clients communicated both through a shared, transactional system and also directly [39]. The clients discovered that isolation caused the state of the shared system to differ depending on the communication channel used. In addition, they were often unable to make progress on requests due to locking imposed by the transactional system.

Therefore, in the approach described in this document we both adopt the concept of rollback and also track dependencies among requests to avoid check-then-act errors when requests roll back. However, we allow developers of the system to decide on the level of isolation that is appropriate for their systems. If developers desire isolation, they can achieve it in our system by adopting two phase locking [43]. Developers may also be able to modify the layout of data and locking protocols to improve performance, using techniques similar to ones being proposed for database systems [95].

Our approach to the problems of locking and rollback is very similar to the concept of open-nested transactions that has been developed for software transactional memory [17, 75]. However, by default the

runtime system takes responsibility for rolling back the changes of redirected requests as necessary. We will discuss the details of our approach in Chapter 5.

2.2.4 Real-time system design

The concept of promptness is closely related to the timing requirements for real-time reactive systems. Approaches for constructing these systems highlight the need for predictable, timely responses to external events. In addition, prior work in real-time systems has dealt with the problems of allocating resources, including the processor, to a stream of prioritized requests.

However, there are several key differences between attentive systems and real-time systems. First, attentive systems often work on time scales that are an order of magnitude larger than the time scales typically addressed in real-time systems. As a result, we propose taking a qualitative approach to execution times rather than setting a quantitative limit, as is typically done in real-time systems. These assumptions are embodied in a calculus of short and long operations that is similar to the O() notation used when discussing algorithmic complexity. We give details of this calculus in Section 2.6.1. This calculus allows us to avoid worst-case execution time analysis [28], which would be difficult to apply given the properties of attentive systems.

Second, in attentive systems the processing times for many requests may not be bounded. Therefore, attentive systems make a distinction between the system's initial response to a request, which is often bounded, and the system's final response to a request, which is often not bounded. As a result, the design of attentive systems must carefully segregate the threads and resources used to identify and redirect requests from other parts of the system, as discussed in Chapter 4.

Third, the designers of attentive systems may not have knowledge of the level of resources that will be available to the system when it runs. The availability of some of the resources, such as network bandwidth, may vary as the system executes.

Finally, unlike real-time systems, attentive systems may depend on subsystems that provide services that are essential to processing their requests. When these subsystems do not directly support attentiveness, it is difficult for the attentive system to make strong guarantees about the processing times of requests. For example, a designer of an email program cannot bound the time to display a message when there is no upper bound on the time that it will take an IMAP server to send the message to the program. Given these constraints, it makes sense for designers to invest in features that will allow the system's client to redirect the system based on its knowledge of resource availability.

We adopt two approaches from real-time system design. First, our metrics for promptness closely follow those created for real-time systems [32]. Second, we adopt the priority inheritance protocol [90] to raise the priority of blocked activities in systems to avoid priority inversion. For example, in an email program we may raise the priority of an ongoing message download that was initiated by the junk mail scanner when the user submits a request to view the same message.

2.3 Promptness

Promptness is one of two major components of attentiveness. **Promptness** is a measurement of the timeliness of the system's responses to requests submitted by its clients. Many systems allow clients to submit new requests at any time, even while the system is processing prior requests. In addition, many systems send multiple responses to each request submitted by their clients. Therefore, we define three different measurements that assess the system's responses for each request: acknowledgment time, processing time, and latency. In some systems clients may be able to detect wait time, the time that the system holds a request without processing it. In other systems wait times are part of processing time. Figure 2.1 is a sketch of how

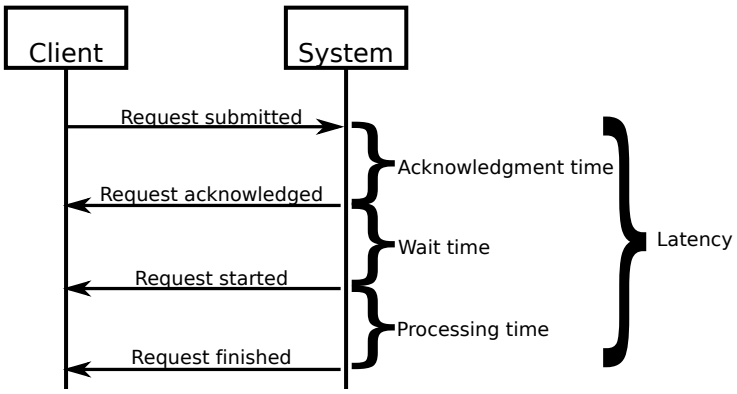


Figure 2.1: Measures of promptness. The times in this figure are specific to a single request. The client is free to submit other requests before the request shown is complete.

these times are measured for a single request. We will discuss each measurement in detail below.

Acknowledgment time measures the delay between the system receiving a request from the client and the system's first, but not necessarily final, response to the request. Clients are sensitive to acknowledgment times because the initial response assures them that the system has received their request. For example, interactive GUI systems should typically respond to inputs, such as clicks of mouse buttons or key presses, within roughly 100 ms [93]. Systems can acknowledge requests by flashing a button, displaying a dialog, or closing a menu. If the system takes longer to respond, the user may assume that the system did not notice the input and repeat it, potentially issuing an unwanted second request. In typical systems the maximum acceptable acknowledgment time is governed by characteristics of the system's clients. Therefore, the acknowledgment times for all types of requests are often identical.

Processing time measures the amount of time that a system spends processing a request. Developers may not be able to bound the processing time for some requests. For example, typical document editors can manipulate documents of arbitrary complexity. Therefore, some requests, such as pasting the contents of the clipboard, can involve an unbounded number of operations. While bounding the processing time for redirection requests is highly desirable, developers may choose to permit unbounded redirection times to achieve greater efficiency. Unbounded redirection times may also reflect factors that are beyond the control of the system's developers, such as communication failures or the presence of inattentive collaborating systems. In many systems the processing time will include some amount of wait time. **Wait time** measures the amount of time that the system holds a request without processing, generally because it would conflict with or consume resources dedicated to a request with higher priority. It is often difficult for clients to distinguish between processing time and wait time.

Latency measures the overall delay between the system receiving a request from the client and the system's final response. Assuming that communication delay between the system and the client is not a significant factor, the latency is the sum of the acknowledgment time and processing times.

The definitions for promptness also apply to sequences of requests. The sequence shown in Table 2.1 is made up of requests and responses between a user and an email program with a folder list, a thread pane, and a message pane. The message pane shows the text of the currently selected message, and the thread pane shows one-line summaries for all of the messages in the current folder. Users can change folders by clicking on the folder list.

Table 2.1 shows a sequence of request: R1, R2, and R3. R1 and R2 display two different messages, M1 and M2 respectively, in the message pane. R3 changes to a different folder, updating both the thread pane and the message pane. The bottom of Table 2.1 shows the values for acknowledgment time, latency, and processing time that would apply if the email program were to execute the requests sequentially.

Time	Actor	Action
T0	User	Issues **R1**: *ViewMessage* M1
T1	System	*Highlights* M1 in the thread pane
T2	System	*Begins to display* M1
T3	User	Issues **R2**: *ViewMessage* M2
T4	System	*Highlights* M2 in the thread pane
T5	User	Issues **R3**: *ChangeToFolder* F2
T6	System	*Displays* M1 in the message pane
T7	System	*Displays* M2 in the message pane
T8	System	*Switches to folder* F2

Request	Ack. time	Proc. time	Latency
R1	T0—T1	T1—T6	T0—T6
R2	T3—T4	T4—T7	T3—T7
R3	T5—T8	T8—T8	T5—T8

Table 2.1: This sequence of requests and responses demonstrates the behavior of an inattentive mail client. Measurements of promptness for each request are given at the bottom of the table.

Time	Actor	Action
T0	User	Issues **R1**: *ViewMessage* M1
T1	System	*Highlights* M1 in the thread pane
T2	System	*Begins to display* M1
T3	User	Issues **R2**: *ViewMessage* M2
T4	System	*Highlights* M2 in the thread pane
T5	User	Issues **R3**: *ChangeToFolder* F2
T6	System	*Switches to folder* F2

Request	Ack. time	Proc. time	Latency
R1	T0—T1	T1-T3	N.A.
R2	T3—T4	T4—T5	N.A.
R3	T5—T6	T6—T6	T5—T6

Table 2.2: This sequence of requests and responses demonstrates the behavior of an attentive mail client. Measurements of promptness are given at the bottom of the table, where applicable. N.A. in the latency column indicates that the request was redirected before completing.

2.4 Behaviors: patterns of redirection

Systems are able to optimize sequences of requests by applying patterns of redirection, called **behaviors**. Optimization relies on both the arrival times of the requests and also knowledge of the request's semantics. An example of a behavior called REPLACE is shown in Table 2.2, which is an optimized version of the sequence shown in Table 2.1. In both tables R2 arrives while R1 is still being processed. This pattern of requests could occur if the user was scanning the thread pane and a large number of messages were displayed. After selecting M1, the user notices M2, a higher priority message. Finally, the user notices M3, not shown in the request stream, which requires him to search for a message in F2.

The arrival of R3, the request to display the contents of F2, makes R1 and R2 obsolete. Their user-visible changes will be undone by the processing of R3. The email program can detect this situation and abandon processing of R1 and R2 when R3 arrives, applying a technique often used in collaborative document editors [82].

In this section we propose a catalog of possible redirections by making several assumptions about the nature of the requests submitted to an attentive system. First, we assume that the system is able to identify two sets of requests: those that are currently active and those that have been submitted but are not yet active. Second, we assume that each part of the system's activity can be attributed to exactly one request. In some systems this latter assumption is not realistic. For example, in systems that employ garbage collection the activity of the garbage collector is due to the sum total of all of the requests, not any individual request. This situation can be handled by assigning a virtual request to these activities.

In an attentive system, redirection begins when a client submits a new request. Table 2.3 defines an attentive system's response to an arriving request in terms of conflicts and priorities. When a request arrives, the system checks it against each of the currently active requests for conflicts. If no conflicts exist, the system ADMITs the request, allowing it to begin processing. The system must not ADMIT a low priority request if it would conflict with one or more high priority requests already in the system. Instead, the system

Priority of Incoming request	Conflicting request		Behavior applied to Incoming request		Conflicting requests
Low or high	None		ADMIT		No change
Low	All low priority	→	BLOCK	*or*	No change
		→	ADMIT		REDIRECT
Low	Some high priority		BLOCK		No change
High	All low priority		ADMIT		REDIRECT
High	Some high priority	→	BLOCK	*or*	No change
		→	ADMIT		REDIRECT

Table 2.3: The behavior of the system is determined by the priorities of requests. When a request arrives, the system constructs a set of conflicting requests. It then chooses an action by comparing the request's priority to the highest priority in the conflicting set. In cases where the priorities are equal, the system has two options, illustrated with the arrows. N.C. indicates that there is no change to the running request. REDIRECT indicates that the system should apply one of the behaviors described in Section 2.4.

should BLOCK the request, delaying it until the conflicting high priority requests complete.

When a high priority request arrives and conflicts with one or more low-priority requests that are already in the system, the system should REDIRECT the low priority requests and ADMIT the high priority request. For example, in most email programs scanning for junk mail is a low priority request[1] while displaying a message is a high priority request. If the system receives a *ViewMessage* request that conflicts with the junk mail scanner, the system should REDIRECT the junk mail scanner and ADMIT the *ViewMessage* request.

Finally, designers must consider cases where the requests with identical priorities conflict. The designers must decide what to do by considering both the system-specific semantics of the requests and also expectations of the system's clients. As mentioned above, a typical email program will treat all *ViewMessage* requests coming from its user as high priority requests. If a new *ViewMessage* request arrives while another one is being processed, it is reasonable for the program to REDIRECT the older *ViewMessage* request to process the new one immediately. This is justified because the user may have made a mistake when issuing the first request or may no longer be interested in the first message. However, if a document editor is busy inserting a character when its user enters a second character, the document editor should BLOCK the second insertion until the first character completes.

When a request arrives, designers can choose one of the following behaviors:

ADMIT allows the new request to begin executing immediately without affecting the other requests in the system. It is appropriate when resources are available and there is no conflict among the incoming requests and the requests already in the system. For example, an email program doing junk mail scanning may choose to **ADMIT** a request to compose a new message.

[1] Here we use the term request quite loosely. The user has configured the email client to detect junk mail, but has probably not issued an explicit request to start scanning. For simplicity in the discussion we are modeling this background task as an implicit low priority request.

Suspend forces one of the requests currently in the system to stop. The client is able to SUSPEND requests to override the system's assignment of priorities to requests. For example, a user may SUSPEND the junk mail scanner in an email program to recover network bandwidth for another system that shares the network. Systems may also SUSPEND one or more requests to implement the other behaviors described below.

Resume allows a SUSPENDed request to continue processing. Like suspend, RESUME may be initiated by either the client or the system. For example, a user of an email program may resume the junk mail scanner when network bandwidth is no longer needed by other systems sharing the network.

Replace forces one or more of the requests in the system to stop executing and allows the incoming request to execute immediately. For example, an email program would be likely to replace an older *ViewMessage* request with an incoming *ViewMessage* request when it cannot display both messages at once. This behavior allows users to recover from slips, such as clicking on the wrong message in the thread pane, with minimal effort.

Interrupt temporarily suspends a request to allow an incoming request of higher priority to execute. Systems use INTERRUPT when either resources are not available to run both requests simultaneously or the requests conflict. First, the system SUSPENDs the low priority request and ADMITs the higher priority request. Once the high-priority request has been ADMITted, the system RESUMEs the low priority request. In some systems clients may be able to invoke INTERRUPT directly, in effect overriding the priorities that the system attaches to requests.

Cancel stops further processing of a request and attempts to undo its effects. It is the equivalent of issuing a SUSPEND followed by an *Undo* request. For example, a user may begin to do a global *SearchAndReplace* of a string in while editing the body of an email. While the *SearchAndReplace* is running, the user may discover a typo in the replacement string. CANCEL allows the user to recover by halting the *Replace* and restoring the original data. In some cases the effects of an operation may not be completely reversible. For example, if a user issues a command to begin writing to write-once media and then cancels the write, the media will be unusable. In this case, the system must send a final response to the CANCEL request that informs its client of the remaining effects of the CANCELed request.

Interface designers specify which of the behaviors described above should be applied to particular sequences of requests. They may want to apply approaches developed for distributed group-ware while doing this work [78]. Their decisions must ensure that the consistency of the interface is preserved from the client's point of view.

2.5 Consistency

While redirection can improve the promptness of systems, clients may be hesitant to redirect systems if it could lead to a system failure. A system failure could occur immediately, causing clients to either lose access to the system or lose prior work done with the system. Failures could also be delayed, creating the risk that one or more clients could lose future work done with the system. For example, canceling a *SearchAndReplace* while editing a message in an email program could corrupt the internal data structures of the system, making it impossible for the client to send the message.

In this section we describe a property called **consistency** that describes a system's ability to provide reliable service. Consistency is simple from the client's point of view: any the system that provides predictable responses to requests and preserves the client's work is consistent. However, consistent systems are difficult to build: developers must preserve the relationships shown in Figure 2.2 to implement a consistent system. The system must preserve relationships among its internal state, its communication with clients, and its communication with collaborating systems. We will discuss each of the forms of consistency below, referring to the labels in the diagram that cover four aspects of consistency: C1, C2, C3, and C4.

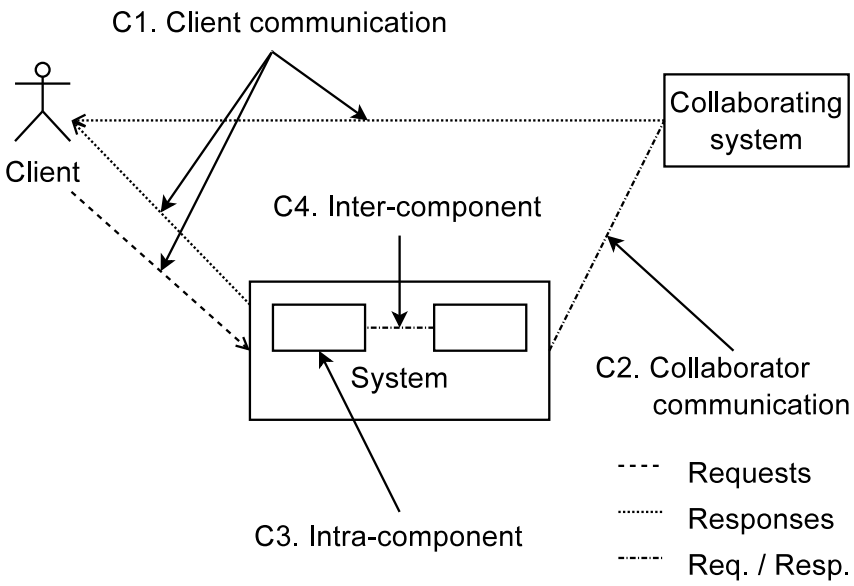


Figure 2.2: Attentive systems must maintain four forms of consistency while redirecting requests.

C1: Consistency of client communication

Client communication can become inconsistent when the system's responses to a request do not match the client's expectations. For example, in many interactive systems the user communicates with the system by sending low-level events, such as key presses and button clicks. The system processes the events and creates a request. In some cases the system's interpretation of the events may not conform to the user's expectations. For example, Android cellular phones allow users to disconnect calls by pressing an "End Call" button on the phone's touch screen. Unfortunately, the "End Call" button transforms into the "Call" button when the other party hangs up. If other party hangs up just as the user is pressing the button, the phone will interpret the press as a request to initiate a new call. In other cases the user can create spurious requests. For example, a user may double click on a link in a browser. Well designed interfaces should ignore spurious requests wherever possible.

Systems can also confuse clients by sending **spurious responses**—responses for requests that have either completed or been redirected by the client. In the example given in Table 2.1, the email program could send a spurious response. In this example the user sends three requests: "Display message M1," "Display message M2," and "Select folder F2." An attentive system may acknowledge the third request by highlighting F2 in the folder list. However, if the system is unable to redirect one of the first two requests, it may display message M1 or message M2 in the message pane. The user, seeing the update, could be confused, assuming that the displayed message is in the highlighted folder. The attentive version of this example shown in Table 2.2 avoids this problem by redirecting the display message requests as soon as the change folder request arrives.

To avoid this failure interface designers must consider the contract between the system and its client when choosing behaviors. In addition, developers may need to create mechanisms to suppress updates from redirected requests to avoid confusing the user.

C2: Consistency with collaborating systems

Collaborating systems may be confused if a system redirects and breaks some invariant of the communication protocol. For example, an attentive system that redirects while communicating with a collaborating system

may transmit a partial request. If the system sends a second request after redirecting, the collaborating system could easily see a malformed request made up of parts of the first and second request.

In addition, attentive systems often must cope with communication protocols that do not allow requests to be redirected after they are submitted. For example, the IMAP protocol both contains long-running requests and also does not allow requests to be redirected. If a user cancels a long-running request, such as a request to move a large number of messages from one folder to another, the email program will be unable to redirect the request promptly.

Many of these problems can be addressed by inserting mediators between an attentive system and collaborating systems to handle redirection. The protocol between the mediator and the attentive system has explicit support for redirection. Mediators avoid partial requests by buffering each request as it is sent, forwarding it to the collaborating system only when the request is complete. The mediator clears the buffer if the system sending the request redirects before the request is complete.

Mediators can often simulate redirecting of the collaborating system for the benefit of the attentive system by modifying the attentive system's requests. In the case of moving a large number of messages, the mediator could transform the move into the following sequence:

1. MARK the messages to be moved with a unique flag
2. COPY the messages from the source mailbox to the destination mailbox. The copy operation will preserve the flag
3. MARK the messages in the source mailbox with the DELETED flag
4. Issue the EXPUNGE command to delete the marked messages from the source folder
5. REMOVE the unique flag from the messages in the destination folder

The mediator can cancel this move operation at any point before step 5. In addition, it can use the unique flag to simulate the effects of canceling the operation while it is still undoing the move on the server.

However, it is important to note that mediators are not a perfect solution. Other systems that connect directly to the IMAP server will be able to observe intermediate states of the copy, including the presence of the unique flag. Some systems, including transactional databases, avoid these problems by isolating partially complete operations. However, this approach has proven to be problematic when applied to interactive systems [39].

C3 and C4: The system's internal state

The system's internal state is made up of both the state within each of the system's components and also the relationships among components. Intra-component consistency describes the state of each component, and can be evaluated against the explicit and implicit invariants of the components. When all of the invariants hold, the entire system has intra-component consistency. In multithreaded attentive systems intra-component consistency can be lost, even in the absence of redirection, due to data races and failures to adhere to the contract specified in the component's interface. For example, it is illegal to modify a `Map`, one of the Java collections, while using an iterator to access its members. If the implementation detects that the system has violated this rule it will throw a `ConcurrentModificationException`.

Redirection, if not carefully coordinated, can also a system to lose intra-component consistency because partially completed changes may be left behind by one of the redirected threads. In many systems, developers cannot be expected to recover the consistency of the system's state after redirection. This scenario is discussed in greater detail in Appendix A.

Inter-component consistency describes the relationships that bind system components together. These relationships are may not be explicitly stated [30], but are often present. For example, in an email pro-

Notation	Length	Description
S	S	Short operation
L	L	Long operation
$S + S$	S	Sequence of two short operations
$S + L$	L	Sequence of two mixed operations
$b * S$	S	Bounded sequence of short operations
$u * S$	L	Unbounded sequence of operations

Table 2.4: Proposed calculus of short and long operations. Short operations are represented with S, long operations are represented by L.

gram components may assume that message identifiers are unambiguous and that a message exists for each identifier mentioned in a folder. The presence of these relationships means that responsibility for preserving and restoring the system's consistency cannot be delegated to the individual components that make up the system. Instead, the design proposed in this work models requests and delegates the responsibility for preserving consistency to the request. This design is discussed in greater detail in Chapter 4.

2.6 Implementing attentiveness

The model that we discussed in Section 2.1 addressed attentiveness from the client's point of view. In this work we adopt a similar model when reasoning about the implementations of systems. However, we expand the model of requests to describe the implementation of systems in general terms. The model contains four operations that control requests: start(), stop(), continue(), and undo(). These operations are sufficient to implement the behaviors mentioned in Section 2.4. In addition, we model dependencies: relationships that form among requests as they execute. Unlike conflicts, dependencies are not apparent in the semantics of the requests. Instead, they are created by implementation decisions that cause requests to share state. Initially, we propose a calculus of short and long operations that allows us to reason about promptness in qualitative terms.

2.6.1 The calculus of short and long: a qualitative approach to promptness

Placing an upper limit on the acknowledgment times of a system may suggest to the reader that all attentive systems are real-time systems. While such an approach is fully compatible with our definition of attentiveness, we believe that developers often find both that this approach is too restrictive for their system and also that a more approximate bound on the acknowledgment times is acceptable.

When this is the case, the requirements for promptness can be defined approximately. For example, in human-system interaction acknowledgment times on the order of 100 ms are often acceptable, while responses of over 1 s can introduce delays that influence the overall task effectiveness of the user [16]. In addition, users may tolerate occasional responses that exceed these limits, especially when doing so gives them greater flexibility in using the system.

We propose a qualitative approach that can be used to reason about promptness: a calculus of short and long operations. This approach assumes that is possible to complete a very large number of low-level operations, such as allocating memory and floating point arithmetic, during the maximum acceptable acknowledgment time for a request. The calculus is described in Table 2.4. In this table, the low-level operations are **short**. For example, individual memory fetches take a fraction of a microsecond, making individual fetches insignificant relative to a 100 ms acknowledgment time. While a hard real-time system would need to consider the potential cost of a page fault, which could raise the memory access time to 1ms or more, most attentive systems are able to assume that page faults will rarely happen.

By applying the calculus, we can determine that entire functions are *short* as long as they involve a bounded number of *short* steps, either individual operations or calls to other *short* functions. For the qsort() function involves a bounded number of operations when the size of the list to be sorted is bounded, and is therefore *short*. However, the number of operations involved in a qsort() of an unbounded list is unbounded, making qsort() long. If developers are unable to determine a bound for the list, they must assert that the function *may* involve an unlimited number of operations and treat qsort() as a **long** function.

It is theoretically possible to achieve attentiveness by constructing a system using only *short* functions. In practice, this is often not possible: many systems contain low-level operations and functions that are not *short*. For example, systems may use network protocols, such as TCP, that have error recovery mechanisms that can block operations for long periods of time. As a result, any methods that invoke operations related to TCP, either directly or by calling other functions, are *long*. Developers must find some way to "contain" the effects of these operations so that the system can check for new requests from its client. While other options exist, such as converting the system to use asynchronous versions of the operations, we suggest design patterns that employ threading to encapsulate *long* operations.

In some cases it is not possible to determine the length of an operation. In most cases, operations with unknown length can be treated as *long* operations in attentive systems. However, the locking operations used to coordinate the execution of multiple threads are a special case. A thread engaged in a *short* operation may need to access state that is shared with other threads, some of which may be engaged in long operations. To preserve the system's consistency the threads may decide to obtain a lock before accessing the state. However, if threads were to hold the lock while executing *long* operations the operation to obtain access to the lock would also be *long*.

It is possible to avoid this problem by attaching additional information to the lock to assert that every thread may engage only in *short* operations while holding the lock. It is reasonable to infer that any thread attempting to acquire access will be blocked only for a short period of time.[2] As a result, it is possible to describe a design pattern that will isolate the concerns of attentiveness from most of the system's implementation. We describe this pattern in more detail in Chapter 4.

The distinction that we draw between short and long operations is well-precedented in GUI toolkits. As we will describe in detail in Chapter 3, the interface between an application and its toolkit is often complex, involving inversion of control, reentrancy, and in some cases recursion between the application and the toolkit's event dispatch system. Since the toolkit is responsible for acknowledging requests, it must regain control within the maximum acknowledgment time specified for the system's requests. However, the requests often involve potentially unbounded computation.

While threading can be used to resolve this problem, prior experience has indicated that deadlocks occur due to reentrant calls when threading is employed within GUI toolkits [52]. In addition, some of the reentrant calls made by the application could affect the processing of future requests.

Most GUI toolkits designate a single thread to process events and detect requests. To protect the consistency of their internal state, they require most calls to the toolkit to be made on this thread. As a result, updates to the internal state are serialized, eliminating the risk of races without incurring the risk of deadlocks associated with locking. To ensure promptness, the toolkit specifies in its documentation that all callbacks from the toolkit to the application must be short. Most developers respond by writing callbacks that assign the long tasks associated with requests to other threads and make any updates needed in the toolkit before returning.

[2] A rigorous proof would require two additional assumptions: the lock is fair when granting access to threads and there are a finite number of threads contending for the lock.

2.6.2 Operations for redirecting requests

To simplify the implementation of the system, we map all of the behaviors mentioned in Section 2.4 to a small number of per-request operations. An architectural element called a *scheduler* uses these operations to implement the behaviors. As a result, a decision on the part of designers to change a behavior for a scenario will not affect most of the system's implementation. In addition, designers are free to invent new behaviors that rely on the same set of operations. The operations include:

start() signals a request to begin processing immediately. This processing generally happens on another thread, which may be created during the start() call.

stop() signals the request, requiring it to stop processing within a short period of time. The request must restore the consistency of the system's state before stopping. Some requests may stop by rolling back some or all of their partially completed work.

continue() signals a request, telling it to continue processing. Once again, the processing happens on a different thread than the thread making the call.

undo() signals a request, telling it to undo as many of its changes to the system's state as possible.

The stop() and undo() operations are asynchronous, returning control to the caller before the operation has completed. In Chapter 4 we propose a signaling mechanisms that requests can use to signal the completion of these operations.

2.6.3 Dependencies

A **dependency** is a one-way relationship between two requests. Our concept of dependency is very similar to the one developed for distributed transaction systems [87]. A dependency between two requests indicates that the first request's processing depends in some way on the completion of the second request. The concept of a dependency between two requests is similar to, but distinct from, a conflict between the same requests.

While an attentive system should always avoid concurrent execution of conflicting requests, it may allow dependencies to form among requests as they execute. In some cases dependencies may be introduced by third-party components without the knowledge of the system's designers. These dependencies are detected and handled by the runtime system described in Chapter 5. Designers of attentive systems must also consider dependencies. In the event of redirection, requests with dependencies must be redirected as a unit, causing a cascading redirection that is analogous to a cascading rollback in database systems. This can greatly increase the cost of redirection. Additionally, designers must determine how much to invest in differentiating conflicts and dependencies. While a clear differentiation can lead to optimizations that greatly reduce the total execution time of sequences of requests, these optimizations come at the expense of coding request semantics into the conflict detection system.

2.7 New requirements introduced by redirection

In the chapters that follow we will define architectural structures and runtime support to help developers to address the following requirements:

The system shall detect and examine incoming requests promptly at all times.

All of the behaviors redirect a system while it is doing other computational work. By default, many systems adopt one of two approaches to requests that arrive while the system is busy: queuing the requests for later processing or dropping requests that arrive while the system is busy. Attentive systems cannot use

these approaches. Instead, they must examine requests as they arrive, possibly redirecting one or more of the requests within the system in response to the new request. Systems can implement this approach by assigning a thread to examine incoming requests. This thread will then delegate the processing of the requests to other threads in the system. Many toolkit-based applications already use this approach. Experience has indicated that developers both struggle to ensure that the delegation is correct [97] and also encounter data races in the implementation of these systems [46].

The system shall be able to redirect any request promptly at any time.

Systems must be able to complete behaviors promptly. Some behaviors, including STOP and CANCEL, specify that the system terminate one or more of the requests that are active when they arrive. Database management systems often offer an administrative interface to kill running transactions. However, this functionality is rarely available in existing systems. POSIX threads and Java threads currently offer APIs that either interrupt threads promptly without preserving consistency or APIs that preserve consistency but may take an unbounded amount of time to interrupt a thread.

The system shall always be able to make progress on incoming requests.

The INTERRUPT, STOP, and SUSPEND behaviors cause their target requests to stop executing without undoing their work. When requests have been suspended while they have exclusive, non-preemptible access to a shared resource it is easy for the system to encounter deadlock conditions. For example, STOP could suspend a thread while it is downloading a message from the IMAP server. A second request that attempted to access the same message has three options:

- Download a second copy of the message, wasting network resources
- Wait for the first request to restart and complete the download, potentially blocking for an unbounded amount of time
- RESUMING the first request, potentially raising its priority until the download is complete

We have adopted the third option. This approach is very similar to the priority inheritance protocol [90] used to resolve inversion of control in real-time systems.

The system shall predict resource competition.

ADMIT specifies that the system should allow multiple requests to run concurrently. However, there is a risk that concurrent requests could cause promptness failures by competing for resources. For example, a *ViewMessage* request in an email program may compete with an attachment download for network resources. However, developers may be unaware of this fact because the email client was implemented with a communication library that hides the network connections. In addition, the system may encounter resource competition from other systems sharing a common network. Attentive systems should, as much as possible, predict internal resource competition and redirect requests based on their priorities to avoid over-subscribing limited resources. This reduces the need for clients to manage the system by redirect requests. However, clients must generally resolve external resource competition manually by redirect requests.

The system shall not ADMIT conflicting requests.

When a system ADMITs multiple requests, it must ensure that its clients can predict the effect of processing these requests in parallel. For example, word processors that allow characters to be inserted into a document while the same document is being printed in the background generally ensure that the changes will not be

reflected in the copy of the document being printed. At a lower level, the UNIX filesystems allow files to be deleted while they are open, while Windows filesystems return errors. Designers, in consultation with the system's clients, must decide on the desired behavior for their systems. These decisions will affect the design of the system's internal state. In addition, implementers may need to cope with conflicts created by third-party components.

The system shall maintain consistency when examining incoming requests while other requests are active.

All behaviors represent some risk to consistency because they require the system to examine requests at all times. We believe that this risk can be greatly reduced by carefully confining the state of the threads that handle incoming requests and request scheduling, ensuring that it is separate from the state used by the system to process the requests. Separating the state allows the thread evaluating new requests to execute without synchronizing with other threads in the system. This approach is well precedented: GUI toolkits also use thread-confined state to avoid consistency problems. Static analysis [97] is now available to verify that implementations conform to this pattern. In addition, we propose runtime checking in Section 5.2 that can verify the separation of state.

The system shall restore consistency after terminating requests.

CANCEL and STOP represent a risk to consistency because they interrupt requests in progress. This is similar to the problems that transactional systems encounter when they abort transactions. However, most transactional systems benefit from the ability to isolate transactions. We explain why we do not believe that this design is appropriate for attentive systems in Section 2.6.3. Developers of attentive systems must also cope with consistency problems that arise when redirecting requests currently executing in third-party components. For example, a user may cancel a request to sort a spreadsheet, and the sorting function may be provided by a third-party component that does not have cancel functionality. We believe that the appropriate response is to use checkpointing as a default implementation, allowing developers to create specialized recovery schemes for specific applications. We discuss our approach in detail in Section 5.1. This is similar to the approach taken in open-nested transactional memory [75].

2.8 Conclusion

The focus of this chapter has been providing a definition of attentiveness, in terms of promptness and consistency, that is testable. Specifically, the definition can be used to evaluate a system's responses to a series of requests to determine if the system's responses were attentive. We have done this by defining the concept of a request and enumerating some of the attributes of requests that allow us to generate a catalog of behaviors that systems can use to improve their attentiveness. To make our definitions concrete, we have examined several examples of requests and responses, contrasting the behavior of attentive and inattentive systems. We have tied our definition to concepts, including short and long operations that can be mapped to code. Finally, we have enumerated problems that developers encounter when building attentive systems.

Later chapters build on the definition of attentiveness given here and explain in greater detail how it motivates the design and implementation of systems. This discussion happens in three parts. In Chapter 3, we define the concept of a directive, a construct that ties the implementation of a system to its design by making specific, often testable, assertions about the system's future behavior. In Chapter 3 we also define a number of directives that can be used in the design and implementation of attentive systems. While directives may be applicable to other problems, our focus in Chapter 3 is on defining directives that allow us to reason about a system's design and implementation in terms of promptness and consistency. We do

this by showing how directives can be applied to specific attentive systems.

In Chapter 4 we first assess the design of multiple systems in terms of attentiveness. We point to specific design features that either support or hinder achieving attentiveness in these systems. Next, we describe a design template that would, in theory, create a highly attentive system. Then we describe a series of experiments where we applied parts of this design to the examples given earlier, assessing the modified systems in terms of attentiveness. We conclude by pointing to issues—redirection of requests, interactions with collaborating systems, detecting dependencies among requests, and checking the accuracy of directives—that point to the need for runtime support.

In Chapter 5 we discuss the implementation of two different runtime systems that we developed to support our experiments. The first runtime system trusts its directives and provides direct support for redirecting requests in single-threaded systems. The second runtime system supports multi-threaded execution and checks directives, but does not provide direct support for redirecting requests. We discuss the implementation decisions that we made when constructing these runtime systems and assess the systems in terms of efficiency and complexity of their code.

Chapter 3

Directives for attentiveness

Attentiveness describes the relationship between the requests received by a system and the system's responses in terms of promptness and consistency. To implement an attentive system, developers must consider promptness and consistency both at the level of design and also at the level of implementation. In this chapter, we propose a class of executable statements called **directives** that allow developers to represent constraints related to promptness and consistency. Directives convey information about multiple aspects of the system's design and its future behavior, such as:

- The relationship between requests and the threads in the system
- Information about regions. Regions are partitions of the system's in-memory state that are assigned to threads as a unit. Regions were first developed for static analysis of concurrency [45], and reduce the effort of reasoning about consistency of systems.
- The relationship between threads and regions, expressed in terms the of **permissions** that threads obtain and relinquish to access regions
- Dependencies among requests
- The maximum acceptable execution time of blocks of code, in terms of the calculus of short and long discussed in Section 2.6.1
- Constraints on redirection imposed by certain pieces of code

The directives described in this chapter do not address all of the state of the system. State that resides outside of the system's memory space and the system's communication with collaborating systems must be managed with system-specific strategies. The runtime system described in Chapter 5 provides a framework to address this state.

Some of the information provided by directives is assertional, describing the system's future behavior. As a result, there is a risk that the information provided by directives will not accurately describe the system's behavior. While some of the information provided by directives can be checked statically [50, 97], in this work we check directives dynamically with the approach outlined in Chapter 5.

Conventional representations of assertional information, such as assertions and invariants, are not well suited to attentive systems. These representations are enforced only at specific program points, such as function calls and returns. Attentive systems rely on constraints that must be enforced *between* program points to check "universal" properties, such as "threads will modify region R only while holding lock L."

Directives, like assertions, apply only to a single thread. The scope of a directive can be defined in abstract terms by using the weak form of the *until* operator [58] in some linear temporal logics. Concretely, a directive is in force from the time that a thread executes the directive until the thread executes a second

directive that overrides it. If the thread never executes such a directive, the first directive remains in force for the lifetime of the thread. A list of all of the directives discussed in this chapter is given in Table 3.1.

Directives provide information both to the runtime system and also to developers. We discuss two variants of the runtime system briefly here and in more detail in Chapter 5: a runtime system implementing trusted execution and a runtime system implementing checked execution. Both runtime systems provide implementations of the four operations needed to support redirection that are described Section 2.6.2: start(), stop(), continue(), and undo(). During trusted execution, the runtime system assumes that the directives in the system's code accurately describe the system's behavior. As a result, the runtime system can be relatively efficient, but may fail to preserve promptness and consistency when directives are inaccurate. The runtime system that implements checked execution verifies that each thread's behavior conforms to the directives that it executes. As a result, checked execution is able to identify inaccurate directives before they can affect the operation of the system.

Below, we discuss both directives and the general model of execution that defines accurate and inaccurate directives. First, we introduce the directives that identify requests and relate them to threads in Section 3.1. These directives allow the runtime system to relate the system's low-level activities, such as modifying memory, to requests submitted by its users.

In Section 3.2 we describe an abstract model of a multi-threaded system. This model represents regions, threads, the permissions that threads have to access regions, permission change events, and accesses to regions. It defines:

- General rules that threads must follow when obtaining and releasing permissions. These rules are enforced in both trusted and checked execution.
- The criteria for determining that a thread's directives are accurate. These rules are enforced only by checked execution.
- Constraints on permission changes needed to avoid data races. The constraints motivate the development of access policies for regions, described in Section 3.3.
- An approach to describing happens-before relationships among requests. This approach, along with directives described in Section 3.3.2, assists the runtime in implementing the undo() operation.

In Section 3.3 we introduce the directives that define regions and attach policies to regions. **Policies** grant and revoke the permissions that allow threads to access regions. Trusted execution assumes that threads access regions only when they have permission to do so. Checked execution enforces permissions by terminating the system when a thread attempts to access a region without permission.

In Section 3.4 we propose an approach for automatically detecting dependencies among requests by using permissions to detect relationships among requests. As we describe in Section 2.2.3, requests are able to observe the partially completed changes of other requests as they execute. These observations create dependencies among requests that can cause check-then-act failures unless groups of requests are redirected as a unit. In some cases developers will need to use additional directives, described in Section 3.4, to inform the runtime system of dependencies that it cannot detect.

In Section 3.5 we propose an approach that developers can use when building systems with components that do not have directives. The approach attaches modifiers to the function signatures that define the interface of these components. During compilation we process these modifiers to create a wrapper for the component called a **tollgate**. The tollgate provides directives for the component to ensure that redirecting requests will not compromise the consistency of the system.

In Section 3.6 we propose directives that document constraints related to promptness. The design of attentive systems often requires that certain threads avoid long operations to maintain the promptness of the system's responses. The runtime system is able to check these directives efficiently, allowing them to be used even during trusted execution.

Requests and threads, Section 3.1
- request_t create_request(bool isShort)
- associate_request(request_t request)
- request_t current_request()
- complete_request(request_t request)
- set_request_priority(request_t request, int priority)
- awaiting_request(request_t r)

Supporting consistency via regions, Section 3.3
- region_t new_region(policy)
- bind(region_t *r)
- associate_global(void *block)
- region_of(block_t *)
- get_ro_slice(? array[], size_t low, size_t high)
- get_rw_slice(? array[], size_t low, size_t high)
- get_transferable()
- get_transferable_ro()
- release_ro_slice(? array[], size_t low, size_t high)
- release_rw_slice(? array[], size_t low, size_t high)
- release_transferable()
- release_transferable_ro()

Support for dependencies, Section 3.4
- no_region_dependencies(region_t *r)
- read_dependency(region_t region)
- write_dependency(region_t region)

Modifiers that define tollgates, Section 3.5
- independent
- reader
- writer
- borrowed_ro
- borrowed_rw
- consumed
- opaque
- transparent

Support for promptness, Section 3.6
- begin_short_section()
- end_short_section()
- short_duration_lock(void *lock)

Special cases for redirection, Section 3.7
- no_rollbacks(region_t *r)
- atomic_sections_are_marked()
- atomic_sections_restore()
- start_atomic() Note: atomic sections are short
- end_atomic()

Table 3.1: This table provides a list of all of the directives that we have defined for attentiveness, referencing the section where they are discussed.

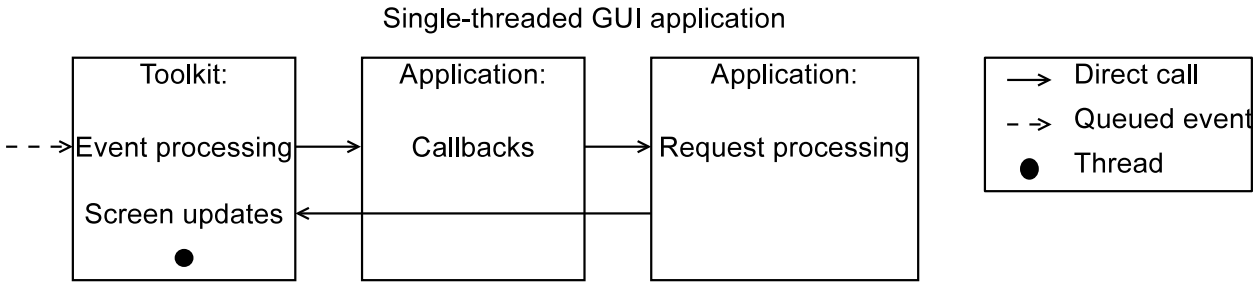


Figure 3.1: A typical design-level view of the relationship between a toolkit and an application.

In Section 3.7 we describe directives that simplify the implementation of the **mediators**: the components that allow attentive systems to communicate with collaborating systems during redirection. To preserve this consistency of this communication, mediators must be able to record information about requests that survives redirection. These directives allow mediators both to create regions that are not modified during redirection and also allow mediators to specify short blocks of code that will execute to completion in the event of redirection.

Finally, in Section 3.8 we apply the directives that describe regions to third-party benchmarks chosen from the PARSEC 2.0 suite [13]. We find that the directives can completely describe the behavior of the benchmarks with an increase in code size of 1%-8%. We will use the same benchmarks when assessing the performance of checked execution in Chapter 5.

3.1 Requests

The design of attentive systems often makes it difficult to perceive the relationship between the system's activities, such as modifying a region, and requests. In this section we describe directives that resolve this problem by both identifying requests and also associating threads with the requests that they are processing.

Some systems must do considerable work before identifying requests. For example, in an application built with a typical GUI toolkit, the toolkit must process a stream of low-level events, such as ButtonPress and MouseMove events, to identify a simple request such as *Paste*. The structure of these applications is shown in Figure 3.1. In this figure a request is identified when the toolkit's thread invokes one of the application's callbacks.

When a thread identifies a request it calls the request_t create_request(bool isShort) directive to notify the runtime system of the new request. The runtime system returns an opaque request identifier that can be used by the application to identify the request in future directives.

The isShort parameter allows developers to notify the runtime system that this request will complete in a bounded period of time, as defined by the calculus of short and long operations given in Section 2.6.1. The runtime system uses this knowledge to reduce the overhead of creating a request. As a result, it will not be possible for the user to redirect the request. The isShort parameter is motivated by prior experience where overhead added to short requests led to attentiveness failures, as described in Section 4.2.1.

In our model of the system each thread in the system is doing work on behalf of some request. When reasoning about threads, we speak of a thread being **bound** to a request. Each thread is bound to at most one request, but one request may be bound to many threads at once. For example, a thread may call a sorting routine that is implemented with multiple threads while processing a request. For the duration of the sort all of the threads used by the sorting routine will also be bound to the request. Each thread identifies the request that it is bound to by calling the associate_request(request_t request) directive. This directive

informs the runtime system that the thread's future actions, such as modifying regions, should be attributed to request. A thread can retrieve its bound request by calling current_request().

When a thread does the final work for a request, it calls complete_request(request_t request). From the thread's point of view, complete_request(request) is equivalent to associate_request(request_t request): the thread will be bound to request, causing its future actions to be associated with request. However, complete_request(request) provides additional information to the runtime system that allows it to free resources. When it executes complete_request(), a thread is asserting:

- No other thread in the system is currently bound to current_request()
- In the future no thread will call associate_request() with current_request()
- In the future no thread will directly request redirection of current_request()

These constraints are created before the thread's bound request is changed to request.

The runtime system should enforce these constraints, treating violations of the first two as fail-stop conditions, since they indicate that the request model provided by the system is inconsistent. The runtime system should handle violations of the third constraint by returning an error from the operation doing the redirection.

Requests in the system have priorities. The set_request_priority(request_t request, int priority) directive changes the priority of the request. Higher values of priority correspond to higher priorities. By default all requests start with a priority of 0. A change in a request's priority may cause one or more requests in the system to be redirected, as described in Section 5.1.

Finally, the awaiting_request(request_t other) directive informs the runtime system that the request bound to the calling thread is waiting for other to complete. This directive is non-blocking, allowing systems to use any appropriate methods to coordinate requests. This directive allows the runtime system to implement a priority inheritance protocol [90], potentially raising the priority of the referenced request to match the priority of the request bound to the thread that executes the directive.

A practical example that illustrates the value of priorities and awaiting_request() is discussed in detail in Section 4.1.3: the junk mail scanner in Thunderbird. The junk mail scanner should usually run as a low-priority task to prevent it from blocking requests directly submitted by the user, such as *ViewMessage* requests. Therefore, some thread should execute set_priority() with a low number on the request that represents the junk mail scanner. As a result, the junk mail scanner will be preempted when the user submits new requests, ensuring that it will not consume network bandwidth needed for these requests. This behavior is implemented by an architectural element called the Scheduler, which is discussed in Section 4.3.1.

However, in some cases the user may submit a *ViewMessage* request for the message currently being downloaded by the junk mail scanner. In Thunderbird 2.0 this creates a redundant message download request, wasting network bandwidth. This problem, which is discussed in more detail in Section 4.1.3, could be avoided through careful design. The junk mail scanner would create a new request for each message download and save a reference to the request.[1] The *ViewMessage* request would check for an active request that was downloading the message, and upon finding it would execute an awaiting_request(request_t message_download) directive to raise the priority of the download in progress. This design would both conserve network bandwidth and also allow the user to benefit from the partially completed work done by the junk mail scanner.

[1] The cache of previously downloaded messages is an obvious candidate.

3.2 Reasoning about consistency and dependencies

In this section we describe the relationship between threads, the consistency of the system, and dependencies among requests. To do this, we rely on the relationship between requests and threads established in Section 3.1. Specifically, we assume that the activities of each thread in the system can be attributed to its bound request. Therefore, an abstract model of the activities of threads can be transformed into an abstract model of the activities of requests.

In our model we reason about threads, regions, the permissions that threads obtain to access regions, and time. Our model is based on models that have been developed in the field of temporal logic [80].

In most models, the memory of the system would be represented as a mapping from an address to a value. However, in our model we are not concerned with the values stored in memory. In addition, we abandon the idea of individual addresses in favor of regions. Therefore, for each read or write to memory we identify the region that corresponds to the address being written and use the region identifier as a proxy for the address. In Chapter 5 we describe a mechanism that ensures that threads always agree on the mapping of addresses to regions. Therefore, in our model we define an unordered set *Regions* that contains all of the regions defined for the system. We also define a variable to refer to an arbitrary region in the rules below:

$r : Regions$

We assume that there are multiple threads in the system, that these threads execute concurrently, and that there is no global view of time among threads. This description of time was originally developed for distributed systems [66]. By adopting it we can greatly improve the efficiency of our runtime support, since creating global consistency is often expensive in multiprocessor systems. In the model we use the unordered set *Threads* as a container for all of the system's threads. We define three variables to refer to arbitrary, but different, threads in the system:

$S, T, U : Threads$

$S \neq T$

$S \neq U$

$T \neq U$

While there is no global view of time in the system, it is possible to reason about the order of actions of a single thread. We define an ordered set, *ThreadTimes*, for this purpose.

$ThreadTimes$

We also define markers to reason about actions for a thread in a pre-determined order. B stands for beginning, and is always the earliest action undertaken by the thread. M stands for middle and is the middle action. E stands for end, and is always the last action in the sequence being discussed. By convention we use uppercase letters to refer to both *Threads* and *ThreadTimes*:

$B, M, E, B', M', E', B'', M'', E'' : ThreadTimes$

$B < M < E$

$B' < M' < E'$

$B'' < M'' < E''$

We define a function τ that holds the history of the actions of every thread in the system *for a particular execution*. In future executions the history of threads may be different. Therefore, τ would normally have a subscript. However, in this version of the model we reason only about a single execution, making the subscript redundant. We have eliminated it for clarity. *Actions* is the set of all possible actions which can be undertaken by a thread. We will define *Actions* in greater detail below:

$\tau : Threads \times ThreadTimes \longrightarrow Actions$

Permissions control a thread's access to regions. A thread may obtain either read-only access to a region or read-write access to the region. During trusted execution a thread should only access the region while it holds permission to do so. If a thread does not follow this rule the runtime system will be unable to detect some of the accesses to regions, potentially leading to consistency failures. During checked execution the runtime system will stop a thread that attempting to access a region without permission before the access occurs. We also define two arbitrary variables to refer to permissions.

$Permissions = \{Read, Write\}$

$p, q : Permissions$

In this model, there are four types of actions:

- A thread can obtain permission to access a region
- A thread can release permission to access a region
- A thread can read a region
- A thread can write a region. Since write access always implies read access, an uninterruptible test and set operation would be represented as a write in this model.

We define a set *Actions* to refer to these events. We also define two variables that refer to arbitrary permissions:

$Actions = \{ Grant(r, t, p), Revoke(r, t, p), ReadR(r, t), WriteR(r, t) \}$

$a, a' : Actions$

So far the model closely follows the implementation of the system. However, it is easier to reason about consistency properties from the perspective of regions. Therefore, we define an ordered set, *RegionTimes*, that describes the order of actions for a particular region. *RegionTimes* cannot be directly compared between two regions, and cannot be directly compared with *ThreadTimes*. This allows us to use this model to reason about systems built with hardware that uses a relaxed ordering of memory operations [2]. By convention we use lowercase letters to refer to *RegionTimes*, and adopt the same variables corresponding to beginning, middle, and end.

$b, m, e : RegionTimes$

$b < m < e$

We use *RegionTimes* to define a function that provides the past history of each region in the system for a particular execution. Like τ, the contents of this function may be different for each execution of the system. Therefore, the function would normally be subscripted. Since we reason only about a single execution below we drop the subscript for simplicity:

$\sigma : Regions \times RegionTimes \longrightarrow Actions$

For readability, we define a short function to determine the region for an action:

$$\forall (a, r) \mid matchesRegion(a, r) \equiv \exists (T, p) \mid a = Grant(r, T, p) \vee a = Release(r, T, p) \vee a = ReadR(r, T) \vee a = WriteR(r, T)$$

The σ function can be derived by applying some simple rules to τ. These rules restrict actions, requiring them to be initiated by the affected thread. This is beneficial both from the standpoint of understanding the model and also for the efficiency of the runtime system, since it allows us to avoid expensive operations to remove permissions from threads. In addition, this restriction allows the actions to establish a correlation

between the thread's timestamp and the region's timestamp, thus allowing regions to propagate dependencies through the system:

$$\forall(T, E, a, r, p) \mid (\tau(T, E) = a \wedge matchesRegion(a, r) \equiv \exists Eb \mid \sigma(r, b) = a$$

The following rule forces the order of a thread's grants and releases to be identical in the σ and τ functions. It is needed because the times in σ and τ cannot be directly compared. While the order of each thread's actions must be preserved, interleaving may cause two adjacent actions for a thread in τ to be separated by the actions of another thread in σ.

$$\forall(T, B, a, r, E, a') \mid (\tau(T, B) = a \wedge matchesRegion(a, r) \wedge \tau(T, E) = a' \wedge matchesRegion(a', r) \implies \exists b, e \mid \sigma(r, b) = a \wedge \sigma(r, e) = a'$$

There are two additional rules that apply to the *Grant* and *Release* actions. These rules are enforced both during trusted execution and also during checked execution. We define these rules with the σ function. First, a thread must release its previous permissions to a region before obtaining new ones:

$$\forall(r, b, T, e, p, q) \mid \sigma(r, b) = Grant(r, T, p) \wedge \sigma(r, e) = Grant(r, T, q) \implies \exists m \mid \sigma(r, m) = Release(r, T, p) \quad (3.1)$$

Second, a thread must obtain permissions to a region before releasing them:

$$\forall(r, e, T, p) \mid \sigma(r, e) = Release(r, T, p) \implies \exists b \mid \sigma(r, b) = Grant(r, T, p) \wedge \forall m \mid \sigma(r, m) \neq Release(r, T, p) \quad (3.2)$$

We can now define the *ReadableRegions* for a thread at a given time in terms of τ. Since, write permissions also grant read permissions, it is sufficient to establish that the thread has some unreleased permission to access the region:

$$\forall(r, T, E) \mid r \in ReadableRegions(T, E) \equiv \exists(B, p) \mid \tau(T, B) = Grant(r, T, p) \wedge \forall M \mid \tau(T, M) \neq Release(r, T, p) \quad (3.3)$$

The definition for *WritableRegions* is similar, but also checks that the permission being held is a write permission:

$$\forall(r, T, E) \mid r \in WritableRegions(T, E) \equiv \exists B \mid \tau(t, B) = Grant(r, T, Write) \wedge \forall M \mid \tau(t, M) \neq Release(r, T, Write) \quad (3.4)$$

Finally, we define the concept of accurate directives. Directives are accurate when each thread in the system reads and writes regions only when it has permission to do so. Checked execution, described in Section 5.2, verifies these properties. If these properties do not hold, our approach of using permissions as a proxy for accesses will be unsound, potentially compromising the integrity of the system during redirection:

$$\forall(T, B, r) : \tau(T, B) = ReadR(r, t) \implies r \in ReadableRegions(t, B)$$

$$\forall(T, B, r) : \tau(T, B) = WriteR(r, t) \implies r \in WriteableRegions(t, B)$$

3.2.1 Dependencies

Dependencies can form among requests processed concurrently in attentive systems, since they are not isolated as they are processed. Normally the runtime would propagate dependencies among requests automatically as threads access and modify regions. For example, consider a system executing three requests, R1, R2, and R3. Each request has a thread that accesses a shared region S. R3's thread modifies S, then R2's thread modifies S, and finally R1's thread reads S.

In our system, we assume that every modification of S is also an observation of S's state. Therefore, at the end of this sequence R2 depends on R3, and R1 depends on both R2 and R3: dependencies are transitive.

Therefore, dependencies can be represented by Lamport Clocks [65] attached to requests and regions. It is important to note that dependencies can be circular: if R3 reads S, it will become dependent on R1 and R2. In other words, the request stream for an attentive system may not be serializable. This is a direct result of the lack of isolation in attentive systems, and can lead to a cascading redirection, where redirecting one request causes a number of other requests to be redirected. We discuss this issue in more detail in Section 2.2.3.

Tracking every access to a region would be prohibitively expensive. However, when the rules described above hold a runtime system can reason about dependencies among threads, and therefore dependencies among requests, without tracking individual accesses to regions. To do this, the runtime system monitors permission changes, using them as proxies for accesses to the region. The runtime system treats every *Grant(Read)* and *Release(Read)* action as a read of the region. It treats every treats *Grant(Write)* and *Release(Write)* as a write to the region.

Next the runtime system uses the model to search for happens-before relationships [66] among the permissions changes of threads. Our approach to establishing a happens-before relationship among threads closely follows the approach given in the Java Memory Model [20]. If the runtime system can establish that a write made by thread S happened before a read or a write made by thread T, then the request bound to T depends on the request that was bound to S when it made the write.[2]

In the abstract model the happens-before relationship is defined as a function:

$$HappensBefore : \langle Threads \times ThreadTimes \rangle \times \langle Threads \times ThreadTimes \rangle \longrightarrow Boolean$$

HappensBefore relationships exist for actions on a single thread:

$$\forall (t, B, E) \mid \langle t, B \rangle \; HappensBefore \, \langle t, E \rangle$$

In addition, happens-before relationships are transitive:

$$\langle S, M \rangle \; HappensBefore \, \langle T, M' \rangle \wedge \langle T, M' \rangle \; HappensBefore \, \langle U, M'' \rangle \implies \langle S, M \rangle \; HappensBefore \, \langle U, M'' \rangle$$

To establish a happens-before relationship among threads it is necessary to relate actions in the σ and τ functions:

$$(\tau(S, M) = a \wedge \tau(T, B') = a' \wedge \sigma(r, b) = a \wedge \sigma(r, m) = a') \implies \langle S, B \rangle \; HappensBefore \, \langle T, M' \rangle$$

We propose an implementation that conforms to this model in Section 5.1.3. The precise happens-before relationships for an execution are somewhat influenced by the interleaving in σ. Therefore, the dependencies for a group of requests may not be identical across repeated executions. However, in the worst case the

[2] It is important to note that S may not exist when T accesses the region. In addition, S may now be bound to a different request.

approximation will be conservative, indicating that a dependency exists where none was encountered in the actual execution. In addition, we assume that permission transfers function as memory barriers in the system. This assumption is sound, even under trusted execution, due to the checking of the sanity of permissions transfers outlined in Invariant 3.1 and Invariant 3.2. Finally, it is important to note that nothing in this model assumes the absence of data races: it is possible for two threads to obtain simultaneous write permissions to the same region or for a thread to obtain read permission to a region while a thread is writing it. To reason about dependencies in these situations we need additional directives, which are described in Section 3.4. We can avoid these directives when developers use policies to restrict the assignment of permissions to threads. These policies also eliminate the risk of undetected data races during checked execution.

3.2.2 Sound policies

Policies are specifications that restrict the assignment of permissions to threads. A policy is either sound or unsound. When threads follow a **sound** policy, the policy limits the assignment of permissions to threads, ensuring that there will be no data races. The policy ensures that for every region there is either a single writer with no readers, or no writers with any number of readers.

Conflicting permissions allow data races, typically by permitting a thread to write to a region currently shared with other threads. Policies that allow conflicting permissions are **unsound**. A system may be free of data races even though it uses unsound policies for some regions. For example, the policy governing access to locks in concurrent systems is unsound, since it allows updates from multiple threads attempting to acquire the lock. However, the implementation of locks avoids data races by using a special class of memory operations that are atomic, reading and updating state in a way that is immune to interference from other threads.

In our abstract model sound policies are governed by two additional constraints. First, write permissions are granted only when every thread has relinquished its permissions to the region:

$$\forall(s,t,r,b,e,p) \mid (\sigma(r,b) = \mathit{Grant}(s,p)) \wedge (\sigma(r,e) = \mathit{Grant}(t,\mathit{Write})) \implies \exists m \mid (\sigma(r,m) = \mathit{Release}(s,p)) \quad (3.5)$$

Second, read permissions are granted only when no thread has write permission to the region:

$$\forall(s,t,r,b,e) \mid (\sigma(r,b) = \mathit{Grant}(s,\mathit{Write})) \wedge (\sigma(r,e) = \mathit{Grant}(t,\mathit{Read})) \implies \exists m \mid (\sigma(r,m) = \mathit{Release}(s,\mathit{Write})) \quad (3.6)$$

We will propose a catalog of policies, most of which are sound, in Section 3.3.

3.2.3 Relating the model to program-specific invariants

Developers typically reason about the correctness of subsystems, including complex data structures, in terms of representation invariants that hold at the start and end of each operation. During the operation, the state of the subsystem may not honor the invariant. When operations are long, it is possible that redirection will interrupt an operation while it is in progress, leading to a consistency failure. An approach to ensuring the consistency of redirection can be expressed by establishing a relationship between invariants and the changes in thread permissions permitted by sound policies. In this section we establish this relationship for a doubly linked list. The implementation of the list is shown in Listing 3.1.

```
IV1   typedef struct node {
IV2     const char *name;
IV3     struct node * prev, *next;
IV4   } Node;

IV6   typedef struct list {
IV7     const char *name;
IV8     Node * head, *tail;
IV9   } List;

IV11  void append(List *list, Node *node) {
IV12    node->next = NULL;
IV13    node->prev = list->tail;
IV14    list->tail = node;
IV15    if (list->head == NULL)
IV16      list->head = node;
IV17    else
IV18      node->prev->next = node;
IV19  }
```

Listing 3.1: The representation invariants for doubly linked list will not hold if this version of the append function is interrupted after IV13 and before IV18.

We define two types of blocks for our example, a doubly linked list. Some blocks are *Nodes*; other blocks are the list head structures:

$n : Nodes$

$l : List$

Finally, we use some "common sense" invariants for doubly linked lists.

$$\forall n \mid ((n.next \neq NULL) \implies (n.next.prev) = n)) \tag{3.7}$$

and

$$\forall n \mid ((n.prev \neq NULL) \implies (n.prev.next) = n)) \tag{3.8}$$

and

$$\forall l \mid l.head \neq NULL \equiv l.tail \neq NULL \tag{3.9}$$

In our example Invariant 3.8 holds until line IV13, and is restored at line IV18. In addition, Invariant 3.9 is potentially relaxed at IV14 and restored at IV16. When a system with only one thread executes this code to completion the invariants there is no way to observe the intermediate states. In effect the invariants appear to hold for the entire execution. However, when systems use multiple threads or support redirection it is possible to observe intermediate states that violate the invariants, potentially causing failures due to the resulting loss of consistency.

To avoid this failure, we apply the concept of permissions outlined in the model to scope the invariants given above for the linked list, creating invariants that are much easier to relate to the system's concrete implementation:

$$\forall(t, M, n) \mid ((n \in \mathit{ReadableRegions}(t, M)) \wedge (n \notin \mathit{WritableRegions}(t, M)) \wedge (n.\mathit{next} \neq \mathit{NULL})) \implies (n.\mathit{next}.\mathit{prev}) = n))$$

$$\forall(t, M, n) \mid ((n \in \mathit{ReadableRegions}(t, M)) \wedge (n \notin \mathit{WritableRegions}(t, M)) \wedge (n.\mathit{prev} \neq \mathit{NULL})) \implies (n.\mathit{prev}.\mathit{next}) = n))$$

$$\forall(t, M, l) \mid ((l \in \mathit{ReadableRegions}(t, M)) \wedge (l \notin \mathit{WritableRegions}(t, M))) \implies (l.\mathit{head} \neq \mathit{NULL} \equiv l.\mathit{tail} \neq \mathit{NULL})$$

These invariants do not restrict a thread's actions when working with writable regions. Therefore, the invariants hold both between operations and also within operations. These invariants ensure that consistency will be maintained as long as permissions changes are governed by sound policies. First, a thread holding write permission to the region containing the data structure must restore the representation invariants before releasing the permission. Second, no other threads can observe the data structure while the writer is making modifications. The runtime system described in Chapter 5 exploits this relationship between policies and invariants to support redirection of operations in progress. It does this by copy the content of the region before each thread gains write permission to the region. In the even of redirection it restores the region from the copy. As a result, all of the invariants for the region are known to hold after redirection.

3.3 Regions and policies

In many systems, developers reason informally about the permission that threads have to access parts of the system's state. A developer may declare constraints such as "lock A protects this array," or "this array is read and written only by thread B." The informal approach adopted by developers to these constraints creates multiple problems:

- The rules governing access to state are rarely documented, and are often not apparent to developers examining the implementation of systems
- The behavior of the system's implementation may not conform to the rules established by developers, leading to intermittent system failures due to consistency errors

In this section we introduce directives to address these problems by relating the informal reasoning of developers to the formal model given above. The first set of directives allows developers to define parts of the system state called **regions**. Each block of memory created by the memory allocations routines is placed into region through a process described in Section 3.3.1. Developers document the constraints described above by choosing an access policy, described in Section 3.3.2, for the region. There are two advantages to developers in doing this work. First, the directives allow checked execution to identify data races that may be present in the system. Second, the directives allow the runtime system to provide the stop(), continue(), and undo() operations that are needed to redirect requests.

3.3.1 Regions

A region is a set of blocks of memory. All of the blocks in a region are treated as a single unit with respect to thread permissions. Specifically, a thread gaining access to one block in a region simultaneously gains

```
S1  typedef struct {int i; char *o; } job;

S3  int main(int argc, char *argv[]) {
S4    pthread_t c;

S6    job *j = malloc(sizeof(*j));
S7    j->i = atoi(argv[1]);
S8    pthread_create(&c, NULL, do_work, j);

S10   ...

S12   pthread_join(c, NULL);
S13   printf("%s\n", j->o);
S14   return 0;
S15 }

S17 void *do_work(void *ctx) {
S18   job *j = ctx;
S19   j->o = malloc(10);
S20   snprintf(j->o, 10, "%d", j->i);
S21   return NULL;
S22 }
```

Listing 3.2: In this example the job allocated at S14 a thread-confined block. The statements in bold type (lines S16 and S20) enforce the thread-confinement via a combination of blocking and happens-before relationships.

access to every other block in the region. A region can be as small as a subset of indexes in an array, called a **slice**. However, in most systems a region is either a single block of memory or a collection of blocks of memory. Many other definitions of regions will do as long as all threads in the system agree on the identity of regions and the regions are non-overlapping.

Our design of regions was originally based on prior work in static analysis tools. This work established that grouping blocks of memory reduces the annotation effort for many systems [44]. While our directives are based on the annotations developed for this work, our directives make a distinction between having a *reference* to a block in a region and having *permission* to access the region. This distinction allows our directives to directly support coding patterns, such as the one described in Listing 3.2, where threads retain references to blocks that they will not access.

The region_t new_region(policy) directive creates a new region. This directive accepts a single parameter that specifies the policy that governs access to the region. The access policy for a region is fixed for the region's lifetime, and controls which *Grant* actions are legal for the region. A list of policies is given in the next section. By definition, the thread invoking the new_region() directive obtains *Write* permissions to the region.

The new_region() directive returns a region identifier that can be used to reference the region in future directives. It is often not necessary to store the region identifier, since it can easily be retrieved by applying the region_of(block) to one of the blocks of memory in the region. However, when the region is first created it contains no blocks, making it necessary to reference the region with its identifier.

The process of populating a region is indirect to accommodate systems that create new blocks of memory in reusable code. Our approach allows the caller of a function to propose a default region for any new blocks that are created, while allowing the function to override the proposed region. Functions should override the caller's region only when allocating blocks completely under their control. For example, a function that involves complex calculations may choose to memoize [73] its results, storing them in encapsulated blocks

guarded by a lock. When allocating the memory for these blocks the function must override the caller's region. However, the function should restore the caller's region before allocating a block used to return the function's result.

For example, consider the char *strdup(char *input) function from the C runtime library, which accepts a string as input and creates a new copy of the string on the heap, returning a pointer to the new string. One caller may use strdup() to allocate a string that will be part of a region that is protected by a lock. Another caller may use strdup() to create an immutable string. Finally, the caller of strdup() may itself be unaware of the policy that will be used to protect the string.

To address these cases, we use define a directive—region_t bind(region_t)—that assigns a "current region" to the thread that executes it. When a memory allocation routine creates a new block, it retrieves the current region for the thread doing the allocation and places the block into it. The current region has no effect the thread's execution outside of the memory allocation routines. Threads are initially bound to a thread-local region when they are created. Since the thread-local policy is the most restrictive, using it as the default eliminates the possibility that races will go undetected due to missing directives.

Defining blocks as by single calls to a memory allocation routine can create problems for some systems. Some systems share state in arrays, assigning permissions to access non-overlapping slices of the array to different threads. For example, the x264 video encoder places frames into a two-dimensional array and grants threads permission to access individual scanlines.[3] We propose specialized directives—get_rw_slice(array, low, high), get_ro_slice(array low, high), release_rw_slice(array, low, high), and release_ro_slice(array, low, high)—that allow threads to request read and write permissions for slices rather than the entire array. Threads are not obligated to use the same ranges in the release directives that they used in get directives. For example, an encoding thread in x264 typically issues a get for the entire array representing a frame, but issues a release for each scanline as it is encoded. Allowing the use of arbitrary, but non-conflicting, ranges in the array slice directives makes it easier to use directives to model the system.

3.3.2 Policies governing access to regions

The assignment of permissions for every region in the system is governed by an access policy. In this section we propose a catalog of access policies, working from the most restrictive policies to the most flexible. In general the most restrictive policies require the smallest number of directives. However, the over use of restrictive policies can introduce attentiveness failures in systems. For example, policies based on locking are sound because they both block threads requesting conflicting permissions and also establish happens-before relationships [65] before they transfer permissions among threads. While these policies are simple to apply and verify, their use of blocking introduces a threat to promptness.

Other policies rely on developers to coordinate the threads of the system to ensure that threads do not attempt to acquire conflicting permissions to regions. To preserve soundness, these policies treat any attempt by a thread to acquire conflicting permissions as a fatal error. It is possible for developers to write code that will fail intermittently when using these policies if the coordination among the threads is inadequate. We discuss this issue in more detail in Section 5.2.3.

Our policies are a superset of the policies described in Chapter 3 of *Java Concurrency in Practice* [40], and are very similar to the types used for checking data sharing strategies in multithreaded C code [3]. They also incorporate knowledge from static analysis for concurrency, which identified patterns of non-lock concurrency [97]:

Thread-local regions can be accessed by only the thread that creates them. Threads do not need to engage in any coordination before accessing thread-local regions. Thread-local regions could be generated as a special case of many other policies. However, we believe that there are several advantages to providing an explicit thread-local policy. First, declaring a region to be thread-local allows a developer reading the code to

[3] x264 represents the frame as a single object, not an array of scanlines.

know that the region will never be shared. In addition, systems with non-uniform memory access (NUMA) can use the information that a region is thread-local to allocate the region in local, fast memory. Finally, we are able to use this knowledge in our runtime systems to make efficient use of memory. Permissions to thread-local regions can never change. Therefore, any thread executing a directive that would result in a *Grant* or *Revoke* action for a thread-local region will stop with an error.

Guarded regions are protected by a concurrency-control construct, defined by the memory model for the system, such as the monitors described in the Java Memory Model [20] or the mutexes described in a memory model being developed for C++ [14]. These constructs provide mutual exclusion by blocking threads. They also create a happens-before relationship between the thread relinquishing the mutual exclusion and a new thread that obtains mutual exclusion.

Developers must point to a specific instance of an appropriate concurrency-control construct, such as a pthread_mutex, when creating a guarded region. The runtime system will automatically process a *Grant*(*Write*) action to the region when a thread obtains mutual exclusion and process a *Release*(*Write*) action to the region when a thread relinquishes mutual exclusion.

Immutable regions go through two phases. In the initial phase the thread that creates the region has exclusive read-write access to it. After initializing the region, the thread will relinquish write access to the region and publish it. In the abstract model, this is equivalent to invoking *Release*(*Write*) followed by a *Grant*(*Read*) for the thread that created the region. Other threads in the system will eventually also receive a *Grant*(*Read*) to the region. The timing of this *Grant*(*Read*) is left to the implementers of the runtime system.[4]

We assume that publishing an immutable region establishes a happens-before relationship between the publisher and any thread that subsequently accesses the region. Other threads must coordinate with the publisher to establish a happens-before relationship before accessing the region. In Java this is called "safe publication" [40], and is often implemented with synchronized blocks. Once a region has been published there is no mechanism that allows a thread to gain write to the region. As a result, there is no way to free the region after it has been published. Therefore, immutable regions should be used only for state that will need to exist for the lifetime of the process that created the region. The phased-immutable policy described below can be used to implement immutable regions that can be destroyed.

Thread-confined regions are read and written by only a single thread, called the owner, at any given time. Unlike thread-local regions, the owner of a thread-confined region may *Release(Write)* it, allowing at most one other thread to claim ownership of the region via a *Grant(Write)* action. An example of thread-confined regions is shown in Listing 3.2. The pthread_create() call at line S8 allows the parent thread to pass a reference to j, the job block created at line S6. The parent must not change the data in j while the child is running to avoid creating a data race. After the pthread_join() call at S12 completes, creating a happens-before edge, the parent can access j to read the results generated by the child thread.

Phased-immutable regions exist in one of three states: exclusive, shared, and unassigned. The phased-immutable policy is the most flexible sound policy in the catalog, giving developers direct control over *Grant()* and *Release()* actions. When they are first allocated, phased immutable regions are in the exclusive state and owned by the thread that created them, which executes a *Grant(Write)* action. The owner can then issue a *Release(Write)* action by executing the release_transferable(region_t) directive, placing the region in the unassigned state. When a region is in the unassigned state any thread can obtain permissions for it. If the thread obtains both read and write permissions by executing the get_transferable(region_t) directive, it issues a *Grant(Write)* action. The thread becomes the new exclusive owner of the region. If the thread obtains only read permissions by executing the get_transferable_ro(region_t) directive, it issues a *Grant(Read)*, causing the region to enters the shared state. In this state no single thread is the owner of the region. When a region is in the shared state additional threads can obtain read permission to it. Threads can *Release()* the read permission that they have obtained by executing the release_transferable_ro(region_t) directive. When the

[4] However, the resulting delay must never result in a false report of a violation of the immutable policy.

last thread releases its read permission, the region moves back to the unassigned state. This definition of the exclusive, shared, and unassigned states ensures that threads obtain only non-conflicting permissions to the region.

Listing 3.3 shows an example of the directives that are needed to create a phased-immutable region containing a single block and transfer the region between a parent thread and a child thread. The parent thread, executing at line A9, first creates a new region governed by the phased-immutable policy, in the process creating a *Grant*(*parent*, *Write*) for the new region. Next, still at A9, the parent thread binds the region. The bind() directive returns the region that was previously bound to the thread, allowing the thread to restore the previous region at line A11. Assuming that the program loader called the main function, the saved region will be the thread-local region for the parent thread.

Line A10 allocates a job block. The runtime system automatically adds the block to the bound region. Since the parent currently has exclusive read-write access to the region, the parent is able to initialize the job block at line A13. The parent then releases its permissions to access the region at line A15, generating a *Release*(*parent*, *Write*).

The release happens indirectly by referring to the job block. We allow developers to get and release regions by referencing one of the blocks in the region rather than referring to the region identifier explicitly. This can be confusing, since the release will apply to every block in the region. However, using blocks as proxies for regions greatly reduces the effort involved in adding directives to existing systems, freeing the developers from the task of storing and forwarding region identifiers. This decision is not fundamental to our approach, and could be easily revised in the future.

Even though the parent thread has released its permissions to access the job block, it has retained a reference to it. The parent can safely pass this reference to the pthread_create() call at A17 because pthread_create() will not use this reference to access the block. Instead, pthread_create() passes the reference as a parameter to the child thread, which starts at line A31. The child must obtain read-write permissions for the region by calling get_transferable() at A32, creating a *Grant*(*child*, *Write*). This call would result in an error if the region if there were an unreleased *Grant*(*Write*). However, a human reader can determine that this will not be the case by examining the code: the release_transferable() at A15 completed before the child was started, and the only other get_transferable() occurs at A21. However, the pthread_join() call at A19 ensures that A21 will not execute until the child exits.

Obtaining permissions for the region gives the child the ability to access blocks within the region, but does not cause new blocks to be placed in the region. Since the child wants to add a new block to the region, it executes bind() at line A33, causing the block at A19 to be placed into the same region as the job block. If the child skipped this binding the new block would have been allocated in the child's thread-local region, causing the dynamic analysis to find a policy violation when the parent accessed the block at A23. The child then releases its permissions at line A39 and exits.

In the parent, the pthread_join() call at line A19 creates a happen-before relationship with the child at the point where the child exits. Therefore, the get_transferable_ro() at line A21 will succeed, granting read permission to the region to the parent thread. The parent thread then reads both the job block and the string added to the job block's region by the child and exits.

Thread-safe regions grant read and write permissions to every thread in the system when they are created. As a result, thread-safe regions are both highly flexible and also unsound. Developers must often assign the thread-safe policy to some regions. For example, pthread_mutex structures must be accessible to every thread in the system to allow threads to use mutexes to coordinate their activity. Allowing this access is safe because mutexes are accessed through special routines, such as pthread_mutex_lock(), that access the region with atomic low-level atomic memory operations that also create happens-before relationships. Similar techniques are used to implement other thread-safe regions, such as non-blocking data structures [57].

Since threads do not use directives to obtain access to thread-safe regions, these regions do not automatically propagate dependencies. Developers can use special directives, described below, to propagate

dependencies in code that accesses these regions.

3.4 Dependencies

We provide several directives to give developers greater control over the propagation of dependencies in the system. Developers must use these directives to propagate dependencies created by thread-safe regions, and may use to gain fine-grained control of dependencies in other types of regions. To do this, they use the read_dependency() and write_dependency() directives.

The read_dependency(region_t region) directive indicates that the request associated with the current thread should inherit the dependencies currently associated with the region. The write_dependency(region_t region) directive indicates that the currently bound request should be added to the region's dependencies.

By default, policies have different effects on dependencies, as described below:

Thread-local regions can create dependencies among requests. Since threads do not need to use directives to request access to their thread-local regions, by default the runtime system assumes that using a thread to process a request makes the request dependent on the thread's local regions. As a result, using a thread to process two requests, one after another, will cause the second request to depend on the first requests. The runtime system will not create these dependencies if developers use the no_region_dependencies() directive. This directive informs the runtime system that the thread's local regions are used only for temporary storage while processing requests and do not propagate information between requests. Grand Central Dispatch [4] places similar restrictions on code that runs in block objects.

Guarded regions normally create a dependency between the requests that obtain access to the region. They assume that every request that gains access to the region modifies the region. Developers can gain greater control over the dependencies at the expense of writing more directives by using the phased-immutable regions described below.

Immutable regions can propagate dependencies among requests. Before the region is published, it will accumulate a dependency on every request bound to the thread that created the region. When other threads obtain read-only access to the region after it has been published every request associated with the thread will become dependent on the dependencies of the region. We expect that developers will normally mark immutable regions with the no_region_dependencies() directive to avoid this behavior.

Thread-confined regions propagate dependencies among requests. The region accumulates dependencies on every request associated with the thread that owns the region. When the region is transferred, the request associated with the thread that obtains the region inherits these dependencies, the set of dependencies associated with the region is cleared, and the currently associated request is added to the set. If the thread still has access to the region when new requests are associated with it, they will be added to the region's set.

Phased-immutable regions propagate dependencies among requests. Threads that request read-only access to phased-immutable regions inherit the set of dependencies from the region, but do not modify the set. Threads that request read-write access to the region inherit the current set of dependencies from the region, and add any requests associated with the thread to the set.

Threads can deactivate the default dependency propagation for these policies for a particular region by executing the no_region_dependencies(region_t region) directive. For example, developers may use this call to deactivate dependency propagation for a sequence counter. The results obtained from sequence counters are rarely affected by the redirection of one of the requests that accessed the counter. Once this directive has been executed on a region developers must issue the directives given above to propagate dependencies manually to avoid consistency failures during redirection.

```
A1  typedef struct {
A2    int i;
A3    char *o;
A4  } job;

A6  int main(int argc, char *argv[]) {
A7    pthread_t c;

A9    region_t saved = bind(new_region(PHASED_IMMUTABLE));
A10   job *j = malloc(sizeof(*j));
A11   bind(saved);

A13   j->i = atoi(argv[1]);

A15   release_transferable(j);

A17   pthread_create(&c, NULL, do_work, j);
A18   ...
A19   pthread_join(c, NULL);

A21   get_transferable_ro(j);

A23   printf("%s\n", j->o);

A25   release_transferable_ro(j);

A27   return 0;
A28 }

A30 void *do_work(void *ctx) {
A31   job *j = ctx;
A32   get_transferable(j);
A33   region_t saved = bind(region_of(j));
A34   j->o = malloc(10);
A35   bind(saved);

A37   snprintf(j->o, 10, "%d", j->i);

A39   release_transferable(tW);

A41   return NULL;
A42 }
```

Listing 3.3: Directives to describe a thread-confined job block, allocated on line A21. The directives in this listing are shown in bold type.

3.5 Tollgates: handling composition

When developers build systems, they often choose to incorporate third party components. Developers may not be able to inspect these components, especially when they are delivered in compiled form. In addition, developers may not have the time or expertise to add directives to these components. This rarely presents a problem for directives related to promptness and requests, since they can usually be placed outside of the module. However, the lack of directives in the module creates risk that the system's consistency could be compromised, either by uncoordinated state sharing within the module or due to inconsistencies introduced during redirection due to a lack of knowledge of the dependencies created within the module.

We address this problem by allowing developers to attach modifiers to the function signatures that define the interface of the module. These modifiers track ownership of the blocks referenced in the signature, permissions to access these blocks, and provide information about the dependencies that may be created by the module. A preprocessor takes this information and creates a **tollgate** for the module. A tollgate is a layer that wraps the module, intercepting calls from the system to the module and returns from the module to the system. The tollgate allows the runtime system to enforce the information provided in the modifiers. Execution passes through the tollgate when one of the functions in the module's interface is called or returns. The tollgate has no effect if both sides of the tollgate have directives or both sides of the tollgate lack directives. In other cases, the tollgate "activates."

In our discussion below we assume that the calling function is part of a module that has been augmented with directives, but that the called function is in an "opaque" module. We believe that the definition of tollgates developed below can also be applied in the opposite situation, a reverse tollgate where the caller has not directives and calls code with directives.

Tollgates do not support the array slices described in Section 3.3. Therefore, in the blocks discussed below are created by calls to the memory allocation routines, such as malloc(). Blocks that remain under the control of the module are placed in a region that is only accessible while the tollgate is active. We call this the **tollgate region**. There is only one tollgate region for the entire system.

When control crosses an active tollgate, the tollgate may reassign ownership of the blocks referenced in the function's parameters and return type. This behavior is controlled by modifiers that are attached to the function signatures that define the tollgate. When blocks are reassigned to the called module, the tollgate moves the blocks into the tollgate region. When blocks are reassigned to the caller, the tollgate places the blocks into the current thread's bound region. Any of the caller's blocks that are not reassigned remain in their original regions. The policies attached to the caller's region will continue to be enforced in functions in the called module during checked execution.

During trusted execution, when policies are not enforced, tollgates act much like the annotations developed for MultiRace [81]. MultiRace checks for race conditions in systems that are composed from third-party components. The annotations used for MultiRace are trusted and predict the read and write sets of the call. Therefore, inaccurate annotations could cause MultiRace to fail to detect a race. A similar problem will occur when the runtime system trusts inaccurate tollgates. Dependencies among requests may be missed, causing consistency failures during redirection.

3.5.1 Syntax of tollgates

In some cases, the tollgate region may impose too many dependencies. For example, in C there are many low-level functions, such as the ones marked with independent in Listing 3.4, that access only the blocks of memory provided in their parameters. Since these blocks are handled by regions, there is no need to create new dependencies due to the call to the library.

We define three modifiers that describe the relationship between functions and the tollgate region:

```
E1  writer int fclose(opaque FILE * fp);
E2  writer int fflush(opaque FILE * fp);
E3  writer opaque FILE * fopen(borrowed_ro const char * filename, borrowed_ro const char * mode);
E4  writer opaque FILE * fopen64(borrowed_ro const char * filename, borrowed_ro const char * mode);
E5  writer int fseek(opaque FILE * fp, long int offset, int whence);
E6  writer int fseeko(opaque FILE * fp, off_t offset, int whence);
E7  writer int fseeko64(opaque FILE * fp, __off64_t offset, int whence);

E9  writer int fgetc(opaque FILE * fp);
E10 writer size_t fread(borrowed_rw void * buf, size_t size, size_t count, opaque FILE * fp);
E11 writer transparent char * fgets(borrowed_rw char * buf, int n, opaque FILE * fp);

E13 writer int fputc(int c, opaque FILE * fp);
E14 writer size_t fwrite(borrowed_ro const void * buf, size_t size, size_t count, opaque FILE * fp);
E15 writer int putchar(int c);
E16 writer int puts(borrowed_ro const char * str);
E17 writer int vfprintf(opaque FILE * s, borrowed_ro const char * format, borrowed_rw __gnuc_va_list ap);
E18 writer int printf(borrowed_ro const char * format, ...);

E20 reader int feof(opaque FILE * fp);
E21 reader long int ftell(opaque FILE * fp);
E22 reader __off_t ftello(opaque FILE * fp);
E23 reader __off64_t ftello64(opaque FILE * fp);

E25 independent void free(consumed void * mem);

E27 independent transparent void * memcpy(borrowed_rw void * dst, borrowed_ro const void * src, size_t len);
E28 independent transparent void * memset(borrowed_rw void * dst, int c, size_t len);

E30 independent accepted char * strdup(borrowed_ro const char * s);
E31 independent int strcasecmp(borrowed_ro const char * s1, borrowed_ro const char * s2);
E32 independent int strcmp(borrowed_ro const char * p1, borrowed_ro const char * p2);
E33 independent transparent char * strcpy(borrowed_rw char * dest, borrowed_ro const char * src);
E34 independent int strncasecmp(borrowed_ro const char * s1, borrowed_ro const char * s2, size_t n);
E35 independent transparent char * strstr(borrowed_ro const char * haystck, borrowed_ro const char * needle);
E36 independent double strtod(borrowed_ro const char * nptr, borrowed_rw char * * endptr);
E37 independent long int strtol(borrowed_ro const char * nptr, borrowed_rw char * * endptr, int base);
```

Listing 3.4: Examples of modifiers from the C runtime library

- Writer indicates that the function obtains a *Grant(Write)* to the tollgate region on entry and does a *Release(Write)* on the region when it exits. The function may both read and write blocks that in the tollgate while it executes. Therefore, the call must propagate dependencies through this region. If no modifier is provided, the tollgate assumes that the function is a writer. For example, fread() is a writer because it modifies the file's buffers, which are under control of the C runtime library.
- Reader indicates that the function obtains *a Grant(Read)* on entry to the function and does a *Release(Read)* on exit. The function may read blocks that are in the tollgate region, but will not modify them. Calling request will become dependent on any writers that have entered the tollgate region before the function exits. For example, feof() is likely to be a reader, since it examines the file buffer. However, if the implementation of feof() memoizes its result, it must be treated as a writer.
- Independent indicates that the function will not access blocks in the tollgate region. Therefore, the tollgate will not create dependencies. For example, the memcpy() routine is likely to be independent because it modifies only the destination buffer.

Developers also add modifiers to each parameter and return value in the function signatures. The modifiers attached to the parameters are shown in Table 3.2.

The accepted modifier indicates that ownership of a block will be transferred from the function to its caller as the function returns. For instance, the return value from the malloc() function would be marked as accepted.

The consumed modifier indicates that ownership of the block will transfer from the caller to the function. For instance, the parameter to the free() function would be marked as `consumed`.

The borrowed_ro and borrowed_rw modifiers apply to the input parameters to functions, to allow the function to borrow [18, 19] read-only or read-write access to the block until it returns. The C runtime library's string copy function, strcpy(destination, source), provides a convenient example of the use of borrowed. The first parameter specifies the destination string, which is borrowed_rw because this string will be written. The second parameter specifies the string to be copied, which is borrowed_ro, indicating that strcpy() will read this block but not write to it. During checked execution the tollgate will check that a thread has obtained the appropriate level of access to the parameters in question.

The borrowed_ro modifier is particularly subtle when callbacks are involved, as is the case in the C runtime library's bsearch(key, array, ..., compare) routine. This routine implements a generic binary search that will work with any sorted array. Since the routine will not modify the array, the key and array parameters are annotated with borrowed_ro. Developers must provide a pointer to a function that can compare two keys in the array when calling bsearch(). This leads to two alternatives when a thread calls bsearch() while it has read-write access to an array. In the conservative case, read-write access will be dropped before control returns to the compare() function. A more pragmatic approach would retain read-write access to the array. We believe that the runtime system should be conservative during checked execution.

The opaque modifier indicates that the caller had no permissions to the block, but grants permissions to the function being called. For example, fopen(path, permissions) returns a pointer to an opaque FILE block to its caller. The caller provides this pointer to other calls, such as fread() and fclose(), to identify the file to be acted on. However, the caller should not examine the file block directly.

The transparent modifier indicates that the caller has permissions to the block, but the function is allowed to hold a reference to the block. Ownership of the block is retained by the caller, and permissions may vary while the called component holds its reference to the block. For example, the key and value parameters to g_hash_table_insert(key, value) in the GLIB library [98] would typically be marked as `transparent` to indicate that the hash table will retain a reference to the blocks while allowing the caller to use the blocks in other data structures.

The tollgate, cooperating with the runtime, can directly check the Reader, Writer, Independent, and opaque modifiers. This is not true for all of the modifiers. The accepted, consumed, borrowed_ro, and

Modifier	Direction	Before call			During call		After Call		
		Owner	Permissions		Permissions		Owner	Permissions	
			Caller	Function	Caller	Function		Caller	Function
`consumed`	A calls U	C	ERW	X	X	RW	F	X	RW
	U calls A	C	RW	X	X	ERW	F	X	P
`accepted`	A calls U	F	X	RW	X	RW	C	P	N
	U calls A	F	X	P	X	P→ERW	C	RW	X
`borrowed_ro`	A calls U	C	R or ERW	X	R	R	C	S	X
	U calls A	C	RW	X	R	R	C	RW	X
`borrowed_rw`	A calls U	C	ERW	X	X	RW	C	ERW	X
	U calls A	C	RW	X	X	ERW	C	RW	X
`opaque`	A calls U	F	N	RW	N	RW	F	N	RW
	U calls A	F	N	P	N	P	F	N	P
`transparent`	A calls U	C	P	X	P	P	C	P	P
	U calls A	C	RW	X	RW	N	C	RW	N
unmodified	A calls U	F	X	RW	X	RW	F	N	RW
	U calls A	F	X	P	X	P	F	N	P
	A calls U	C	P	N	P	P	C	P	N
	U calls A	C	RW	N	RW	N	C	RW	N

block owner is indicated by:

- C block owned by the caller's module
- F block owned by the function's module

Permissions

- E Thread has exclusive access
- N Reference with no permissions
- P Permissions determined by the block's policy
- R Read permission
- S Same as before call
- W Write permission
- X No reference

Module status

- A Annotated
- U Unannotated

Table 3.2: Definition of modifiers that define tollgates. The accepted and consumed modifiers shown above the break are sufficient for compositional checking. The modifiers below the break provide more complete documentation of the module's interface. The terms caller and function refer to the caller and function's modules. The caller column in the during call section describes the access granted to blocks to other threads in the calling module for the duration of the call. Unmodified is a special case, applying only to defective tollgates, indicating the permission changes for blocks that cross the tollgate without a modifier.

borrowed_rw modifiers describe the presence or absence of references to regions. The tollgate and runtime have no way to directly check these directives: they check accesses to regions rather than tracking references to regions. Instead, the tollgate translates these directives to permissions changes that continue to be enforced after execution leaves the tollgate. As a result, any attempt to use an invalid reference will create an error.

We have created a partial tollgate for the C runtime library. The directives needed to create this tollgate are shown in Listing 3.4.

3.5.2 Blocks allocated behind a tollgate

The accepted modifier allows the caller to receive blocks that were initially allocated by the function behind the tollgate, called the callee below. When this modifier is absent blocks allocated by the callee can be placed into the tollgate region immediately. In this case, the presence of the memory allocation makes the callee a writer of the tollgate region. Below we will discuss two cases that occur during checked execution: handling new blocks when the function is marked as a writer and handling new blocks when the function is marked as either a reader or independent.

Writers

When a writer allocates a new block, the block is logically part of the tollgate region. However, placing the block into the tollgate region immediately complicates the implementation of the accepted keyword, since the tollgate will need to obtain exclusive access to the block while moving it to the caller's bound region. This would involve forcing the other threads currently behind the tollgate to issue *Release(Write)* actions before the caller can exit the tollgate. Forcing this level of synchronization among threads is very expensive and would greatly increase the expense of exiting tollgates.

Instead, writers place each new block into a phased-immutable region. If a second thread attempts to read or write the block, it will encounter an error. However, in this case the error does not indicate an inconsistency in the directives, since the new block was logically in the tollgate region. The second thread simply does a *Grant(Write)* on the block and resumes execution.

When the first thread begins to exit from the tollgate, it identifies the blocks that correspond to accepted keywords. It then examines the σ functions for these blocks. If the σ function indicates that no other threads have accessed the block, the first thread is able to reassign the block to the caller's region immediately. If the σ function indicates that other threads have accessed the block, the first thread contacts these threads, forcing them to issue a *Release(Write)* for the region. When this process is complete, the region can be reassigned. Any blocks allocated behind the tollgate that are not reassigned are added to the tollgate region.

Reader or independent

Some functions that allocate blocks are not writers. For example, accepted char * strdup(char *string) allocates a block of memory, copies string into the block, and returns the block to its caller. It is highly unlikely that strdup() will access any blocks in the tollgate region. Therefore, making strdup() a writer would unnecessarily propagate dependencies through the system.

To handle this case, we use a special tollgate for functions that have the accepted modifier but do not have the writer modifier. In these functions, we create a phased-immutable region for every memory allocation, placing the newly created block into the region. The creation of the region issues a *Grant(Write)* on the region to the thread executing behind the tollgate.

If a second thread, also executing behind the tollgate, attempts to read from or write to the block, we report an error. The access by the second thread indicates that the first thread acted as a writer, making the

modifier that defined the tollgate inconsistent. A developer can resolve this error by modifying the keyword to be writer. In some cases it is possible to resume the execution of the system by imposing the writer keyword at runtime.

Assuming that the function executes to completion, the tollgate processes the accepted keyword, assigning a new policy to the referenced blocks based on the bound region of the calling thread. Any blocks that remain are not added to the tollgate region, since this would make the function a writer. These blocks are owned exclusively by the thread that allocated them, but are accessible only when the thread is behind the tollgate. Therefore the tollgate will issue a *Release(Write)* on these regions as it exits and issue a *Grant(Write)* on these regions when it reenters.

3.5.3 Rules governing the implementation of tollgates

We have adopted the following rules in our design and implementation of tollgates. These rules are designed to make the effect of tollgates clear and to minimize the chance that consistency errors can arise as execution moves through tollgates.

Every block in the system is owned by either the caller or the called module, but never both. For convenience we call these blocks caller blocks and callee blocks. This rule has two implications. First, it specifies that there are no blocks in the system that are owned by neither the caller nor the callee. Second, it asserts that at any given point in time all of the threads in a system agree on the owner of any given block. If this were not the case threads running in a module without directives could possibly update a block after it passed through a tollgate to the system, causing undetected consistency failures in blocks protected by sound policies even under checked execution. Catching these errors during checked execution allows developers to detect and resolve inconsistencies in the tollgate for the module in question.

Every access to a caller block is checked. This rule ensures that modules without directives are not able to violate the access policy put in place by the caller during checked execution. These violations indicate that the tollgate for the module does not accurately reflect the module's behavior.

A callee block may become a caller block only when a thread crosses an active tollgate. The block must be referenced directly or indirectly by the annotated signature. The tollgate will assign a policy to the block, taking the policy either from the modifiers in the function's signature or the policy bound to the thread crossing the tollgate. The thread crossing the tollgate will gain exclusive access to the block before exiting the tollgate. This rule governs the behavior of the `accepted` modifier, and ensures that callee blocks cannot become caller blocks spontaneously.

A caller block can become a callee block only when a thread crosses a tollgate. The block must be referenced directly or indirectly in the annotated signature. The caller must obtain exclusive access to the block before entering the tollgate. The block may not be thread-local[5] or immutable. We have used this rule only for routines that either destroy or relinquish control of blocks, such as free() and fclose(). This rule governs the behavior of the `consumed` modifier. Like the previous rule, it ensures that blocks cannot become callee blocks spontaneously. It also places responsibility on the caller for ensuring exclusive access to blocks crossing a tollgate. By forcing the caller to obtain exclusive access, we ensure that no other thread will access the block as it crosses the tollgate. If these accesses were allowed they may not be checked against the block's policy, potentially causing false negatives.

A thread may access a block within a tollgate, but the accesses must occur while the block is a caller block. Typically blocks are not accessed within tollgates. However, when a collection of blocks passes through a tollgate and the ownership of the collection is transferred, the tollgate may need to read fields in one or more of the blocks to identify other blocks that must be transferred. For example, if a tree is passed through a tollgate the tollgate would need to read the left and right pointers of each node to identify nodes in the tree. We specify that these accesses must happen while the block is protected by a policy. As

[5] Functions that destroy blocks, such as free(), are treated as a special case. See Section Section 5.3.3.

```
I1  void call_annotated_module(consumed char *data);

I3  char *blocks[3];
I4  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
I5  int locked_buf = 0;

I7  void *do_work(void *ctx) {
I8      int id = ctx;
I9      char *obj;

I11     pthread_mutex_lock(&lock);
I12     obj = blocks[locked_buf];
I13     locked_buf = id;
I14     pthread_mutex_unlock(&lock);
I15     call_annotated_module(obj);
I16     return NULL;
I17 }

I19 int main(int argc, char *argv[]) {
I20     pthread_t c1, c2;

I22     blocks[0] = malloc(1);
I23     blocks[1] = malloc(1);
I24     blocks[2] = malloc(1);
I25     pthread_create(&c1, NULL, do_work, 1);
I26     pthread_create(&c2, NULL, do_work, 2);
I27     pthread_join(c1, NULL);
I28     pthread_join(c2, NULL);
I29     return 0;
I30 }
```

Listing 3.5: This is an example of race-free code that is non-deterministic. block[0] will always be passed to the annotated module in this code. However, the second block will either be block[1] or block[2], depending on the interleaving of the two child threads. A race could go undetected if developers attempt to compose multiple runs of the code that pass different blocks to the tollgate. The locked section from I11-I14 ensures that there is no race between the threads that attempt to allocate blocks.

a result, checked execution will detect any races that could occur as the block passes through the tollgate. These accesses indicate that the ownership transfer specified in the tollgate is not sound, since threads in both the caller and the callee are attempting to access the block.

When combined, these rules allow us to define a set of checked accesses for each thread. Each thread's set of accesses depends solely on its activity and the identify of blocks entering tollgates. As a result, the content of each thread's set is completely independent of thread interleaving.

During checked execution the runtime will not generate false positives, since it checks only accesses that affect blocks currently governed by a policy. Therefore, every error generated by the system can point to directives that established a policy for the block, the permissions for the thread attempting to access the block, and the access that violates the policy. Developers can then trace each of these reports to a defect in the system's implementation, one or more directives, or one or more tollgates.

We can trivially demonstrate that the runtime system avoids false negatives during checked execution when all of the modules in a system have directives. However, in systems with active tollgates it is possible to create intermittent false negatives by manipulating the identity of blocks passed through tollgates. An example of code that does this is shown in Listing 3.5.

3.6 Promptness

Promptness is also a concern for attentive systems. Therefore, we have developed directives to allow developers to document their assumptions about promptness in implemented systems. Our approach to promptness differs from our approach to consistency. First, there is some ambiguity in the definition of promptness that we propose. The calculus of short and long operations that we propose in Section 2.3 focuses on bounded and unbounded sequences of operations. However, from the client's point of view, promptness is generally determined by measuring the execution time of sequences in terms of acknowledgment times. Our directives of promptness are based on the calculus of short and long operations. However, the runtime system checks the directives with reference to the acknowledgment time.

Second, we do not treat a promptness failure as a fail-stop condition. Instead, promptness failures should be logged. The log entry should point to the directives that were violated. Ideally the entry would also include the call stack.

There are several reasons to avoid treating promptness failures as a fail-stop condition. First, unlike consistency failures, it is possible to allow a system to continue executing after a promptness failure without creating an additional threat to attentiveness. Second, runtime systems are often able to address promptness failures by falling back to more conservative forms of redirection that ignore the inconsistent directives. Finally, the design of many systems makes it much more difficult to be confident that directives related to promptness will be accurate. Logging rather than stopping allows developers to be aggressive in adding and adjusting these directives, thus allowing them to gain knowledge about their systems quickly. In addition, it allows them to leave checking in place in deployed systems, potentially providing debugging information for attentiveness failures in these systems.

Below we will describe three promptness directives. The first allows developers to mark a sequence of operations that they believe will be short. The second allows developers to inform the runtime system that a short sequence of operations should be atomic with respect to the STOP operation. We will demonstrate that these annotations allow developers to create algorithms that can be STOPped while retaining partial changes by applying them to a sorting algorithm. Finally, we will describe annotations that allow developers to identify short-duration locks [43]. Short duration locks allow developers to create systems that maintain consistency and promptness while sharing data between threads engaged in short sequences and threads engaged in long sequences.

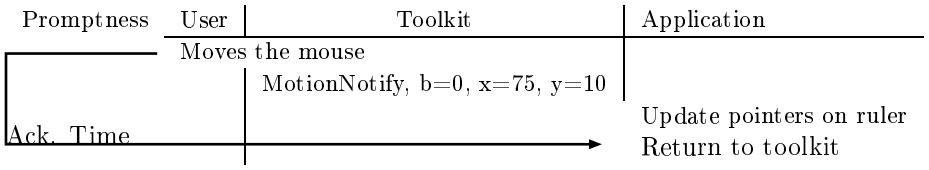


Table 3.3: The toolkit and application must cooperate when processing events for simple mouse movements. The promptness of the system depends on the application returning within the acknowledgment time.

3.6.1 Short sections

The need for short sections becomes apparent when we examine Figure 3.1, which shows an abstract model of a toolkit based application. The arrows in this model show the path normally taken by requests in the application. The application shown here uses only a single thread, and is therefore very vulnerable to promptness failures.

When viewed from the perspective of attentiveness, the system receives a stream of events from the user, uses the toolkit to interpret these events and create a request, and finally passes the request to the application-specific code via a callback. The abstract path of this execution is shown in Table 3.3. Since the toolkit is single threaded, shares a thread with the callback, and must execute before requests can enter the system, the promptness of the entire system depends on this callback returning within the acknowledgment time.

For example, the model given above corresponds to the architecture of Inkscape, one of our case studies. Inkscape is a vector graphics editor and does not bound the complexity of documents. Therefore, users can produce promptness failures in Inkscape by running commands on complex documents. For instance, a user doing a *SelectAll,* followed by a Copy, followed by a *Paste* can cause the user interface of Inkscape to lock for several minutes.

While careful use of threading could reduce the risk of an attentiveness failure, threading cannot be applied arbitrarily to this system. Specifically, most GUI toolkits require that the code in the toolkit and the application's callbacks be executed by a single thread. This constraint allows toolkit designers to avoid the risk of deadlocks and inconsistencies that occur in multi-threaded toolkits due to reentrant calls from threads [52].

Listing 3.6 shows directives that encode this requirement. This code is taken from the GTK+ toolkit. We chose to use the gtk_propagate_event() function because it is the last function within the toolkit that is guaranteed to be on the call stack of every application callback invoked by the toolkit. The begin_short_section() directive indicates that execution must reach the end_short_section() directive well within the acknowledgment time for the application. These directives can be nested. The timing constraint will always apply to the outermost pair of directives. However, when reporting inconsistencies the runtime system should report every short section that violated the constraint. By providing this information, the runtime system allows developers to use nested short sections to diagnose the underlying cause for a promptness failure in their application while allowing toolkit developers to place directives aggressively to highlight all of the promptness failures caused by defective callbacks.

This example also illustrates the need to place these directives speculatively. Mouse movement is a very simple operation. In fact, the callback registered by Inkscape needs only to update two markers that run along the horizontal and vertical rulers displayed at the sides of the document to reflect the new position of the mouse. Therefore, it is fairly clear, based on the semantics of the request, that this call will be short. However, at the level of implementation this call involves the complex call graph shown in Figure 3.2. This call graph has been simplified so that it shows only calls that cross between Inkscape and the libraries that it uses, including the GTK+ toolkit. Functions are shown as boxes, and four boxes have been highlighted.

```
EV23  /**
EV24   * gtk_propagate_event:
EV25   * @widget: a #GtkWidget
EV26   * @event: an event
EV27   *
EV28   * Sends an event to a widget, propagating the event to parent widgets
EV29   * if the event remains unhandled. Events received by GTK+ from GDK
EV30   * normally begin in gtk_main_do_event(). Depending on the type of
EV31   ...
EV32   *
EV33   **/
EV34  void
EV35  gtk_propagate_event (GtkWidget *widget, GdkEvent *event) {
EV36      ...
EV37      begin_short_section();
EV38      if ((event->type == GDK_KEY_PRESS) || (event->type == GDK_KEY_RELEASE)) {
EV39          ...
EV40              handled_event = gtk_widget_event (widget, event);
EV41          ...
EV42      }

EV44      /* Other events get propagated up the widget tree
EV45         * so that parents can see the button and motion
EV46         * events of the children.
EV47         */
EV48      if (!handled_event) {
EV49            while (TRUE) {
EV50          ...
EV51                    handled_event = gtk_widget_event (widget, event);
EV52                    if (!handled_event && widget)
EV53                        g_object_ref (widget);
EV54                    else
EV55                        break;
EV56            }
EV57      }
EV58      end_short_section();
EV59  ...
```

Listing 3.6: Directives to express the promptness requirements for callbacks in a general way from within the toolkit.

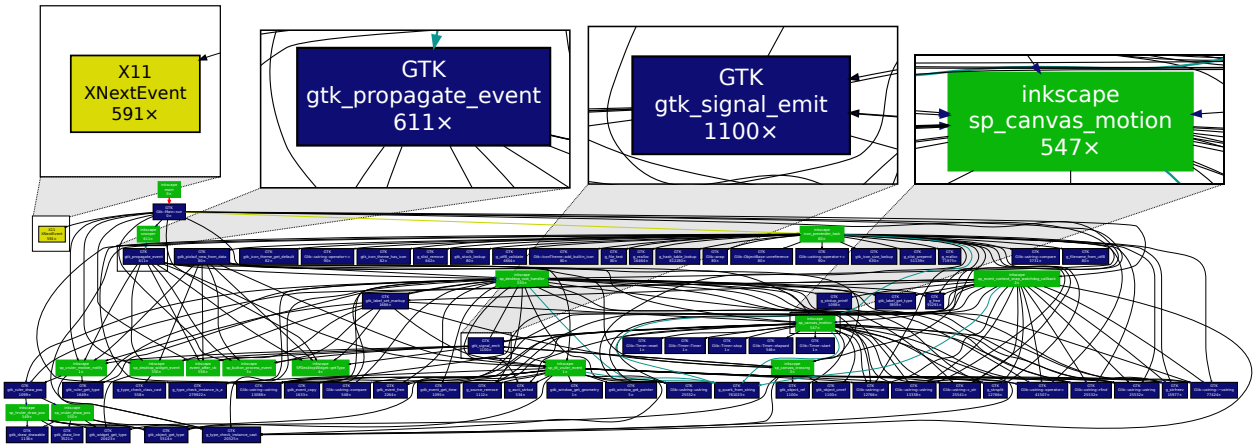


Figure 3.2: Interactions between the toolkit and application are complex. This is a simplified call graph showing calls between Inkscape and its toolkit for a mouse movement. The document is not changed as a result of the movement.

The GTK+ toolkit uses XNextEvent() to received new events from the user. These events are processed, eventually resulting in a call to gtk_propagate_event(). The gtk_signal_emit() call invokes a callback that handles the event, registered by either the application or by GTK+. Finally, the sp_canvas_motion() callback in Inkscape is the first point in the code where we can identify the semantics of the request in the code.

The numbers shown at the bottom of the highlighted boxes indicate the number of times that a function is called. Note that there are approximately two gtk_signal_event() calls for every sp_canvas_motion() call, but that the number of sp_canvas_motion() calls is approximately the same as the number of XNextEvent() calls. This, along with the arrows entering gtk_signal_event(), suggests that code in Inkscape forms a second event, and re-dispatches it through the GTK+ toolkit. Indeed, preliminary analysis of the call graph indicates that each mouse movement causes control to cross the Inkscape-GTK+ boundary 201 times. Notable examples include:

1. GTK+ calls Inkscape with the mouse-moved event
2. Inkscape calls GTK+ to update arrows on the rulers showing the mouse position
3. GTK+ calls Inkscape's second mouse-motion callback
4. Inkscape calls the toolkit to emit signals to other widgets
5. GTK+ dispatches some of these events to Inkscape's custom ruler widgets
6. Inkscape's ruler widgets call inherited methods in GTK+
7. GTK+ calls overridden drawing methods in Inkscape's ruler widgets
8. Inkscape calls low-level drawing primitives in GTK+

This complexity is not specific to Inkscape: researchers have discovered that developers struggle to answer similar control-flow questions about other systems [68, 1]. In light of this, our runtime system does not treat inconsistencies in promptness directives as a fail-stop condition. Instead, it logs the inconsistency and allows computation to proceed. In some cases the runtime system may not be able to redirect the requests in question. In other cases it will fall back to a more conservative, but also more expensive, form of redirection such as rolling back to a checkpoint taken before the request was started. Redirection is discussed in greater detail in Section 5.1.

3.6.2 Duration of blocking

The blocking associated with guarded regions can present a threat to promptness. To allow threads engaged in short operations, such as those involved in acknowledging requests, to use locks, we allow developers to identify short duration locks [43]. These locks can be held only for short durations of time.

Developers use the short_duration_lock(void *lock) directive to indicate that a lock is only held for short periods of time. Once this directive has been issued for a lock it cannot be removed. Issuing the directive multiple times for the same lock does not indicate an error in the system's model.

During checked execution the runtime system will log an inconsistency when threads attempt to acquire locks that have not been marked as short duration locks while executing in short sections of code. In addition, the runtime will log cases where a thread holds a short duration lock for a long period of time as an inconsistency.

3.7 Redirection and mediators

In this section we describe the directives that offer an alternative to assigning the responsibility for managing consistency to the runtime system. Normally, the runtime system ensures the consistency of the system's state during redirection by rolling back the contents of regions. This approach helps developers to incorporate third-party code that does not directly support redirection by removing all evidence of the redirected request from the system's state. However, the runtime cannot resolve every consistency failure. For example, a system may initiate communication with a collaborating system, such as an IMAP server, on behalf of a request. If the request is later rolled back, the runtime has no way of undoing the effects of the communication. Developers handle this problem by implementing mediators—components that coordinate redirection with collaborating systems. Mediators must retain information about redirected requests to compensate for their effects in future communication. To implement a mediator developers must:

- Identify the state that must be preserved
- Maintain the consistency of the state
- Allow timely redirection
- Address problems that can occur when code is shared by mediators and other parts of the system

In this section we describe directives that allow developers to address all of these problems. We will use the qsort() function shown in Listing 3.7 to discuss atomic sections. The qsort() function is *long*, varying between $O(n \log(n))$ and $O(n^2)$ where n is the number of elements in the array. This function has been identified as the cause of attentiveness failures in a CD database [31]. We will describe the implementation of this function in detail below, after using its interface to motivate the directives that support atomic sections.

Developers can identify the state that must be preserved by placing it into regions and identifying these regions to the runtime system. To identify the regions, threads execute the no_rollbacks(region_t r) directive, informing the runtime system that the region should not be modified during redirection.

There is a risk that a request could leave inconsistent changes in the region when it rolls back. To avoid problem, developers use directives to delay rollback for a short period of time as they update the region. These directives create **atomic** sequences of code that will always execute to completion once they begin to execute. The presence of atomic sequences delays redirection, potentially introducing a risk to promptness. Therefore, atomic sequences must also be short sequences. Threads identify the beginning of an atomic sequence by executing the start_atomic() directive and execute the end_atomic() directive to mark the end of a sequence. It is possible to nest these directives. When this happens, the outermost pair of directives defines the atomic section.

```
Q1  static void swap(char * restrict a, char * restrict b, size_t sz) {
Q2      start_atomic();
Q3      while(sz--) {
Q4          char t=*a;
Q5          *a++ = *b;
Q6          *b++ = t;
Q7      }
Q8      end_atomic();
Q9  }

Q11 static inline int atomic_compare(int (*compare)(const void *, const void *), const void *a, const void *b) {
Q12         start_atomic();
Q13         int rval = compare(a, b);
Q14         end_atomic();
Q15         return rval;
Q16 }

Q18 static inline void qsort_inner(char *base, size_t nel, size_t width, int (*compare)(const void *, const void *)) {
Q19     if ((nel == 2) && (atomic_compare(compare, base, base + width) > 0))
Q20             swap(base, base + width, width);
Q21     if (nel < 3)
Q22         return;
Q23     char *left = base;
Q24     char *right = base + nel * width;
Q25     while(left + width != right) {
Q26         while ((left + width != right) && atomic_compare(compare, left + width, base) <= 0)
Q27             left += width;
Q28         while ((left != right - width) && atomic_compare(compare, base, right - width) <= 0)
Q29             right -= width;
Q30         if (left + width == right)
Q31             break;
Q32         swap(left + width, right - width, width);
Q33         left += width;
Q34         right -= width;
Q35     }
Q36     swap(base, left, width);
Q37     qsort_inner(base, (left - base) / width, width, compare);
Q38     qsort_inner(right, (base + nel * width - right) / width, width, compare);
Q39 }

Q41 void gsh_qsort(void *base, size_t nel, size_t width, int (*compare)(const void *, const void *)) {
Q42     atomic_sections_are_marked();
Q43     qsort_inner((char *)base, nel, width, compare);
Q44     atomic_sections_restore();
Q45 }
```

Listing 3.7: An implementation of the qsort() routine with atomic blocks. The routine can be STOPped in a short period assuming that compare() is short and the size of elements is bounded. When STOPped, the list will be partially sorted, but all of the elements will be present. The directives added to the code are shown in bold type.

It is difficult to apply atomic sections to code that is shared between mediators and other parts of the system. The quick sort function, qsort() provides a model of these problems. The function's signature accepts an unbounded[6] number of elements, an unbounded element size, and a pointer to a comparison function called compare(). The current implementation places the compare() function in an atomic section, but cannot be certain that compare() will return in a bounded amount of time. Since the compare function is provided by the caller, the developer of qsort() cannot ensure that the function's execution time will be bounded. To resolve this problem, we require the caller to activate the atomic sections by executing the atomic_sections_are_marked() directive. In executing this directive, the caller is making multiple commitments:

- The compare() function is short
- The size of the elements in the array is bounded, making swap() short
- The caller is willing to accept a reordered, but still complete, array if a request is redirected within qsort()

Before returning from the mediator, the caller issues the atomic_sections_restore() directive. Assuming that there is no nesting, this has the effect of deactivating the atomic sequences, making the start_atomic() and start_atomic() directives embedded in qsort() noops.

It is possible to stop qsort() promptly, leaving the list partially sorted, when the caller is willing to make the commitments outlined above. There are two risks to consistency in the implementation of qsort() shown in Listing 3.7:

Stopping swap() may corrupt the entries being exchanged. This happens because swap() modifies the array provided by the caller. This risk is resolved by placing a start_atomic() at Q2, the beginning of the swap() function and placing an end_atomic() at Q8. This atomic block is known to be short because any caller that activates the atomic section commits to bounding the size of the element.

Stopping compare() could leave inconsistencies in the system's state if the function has side-effects.[7] This risk is resolved by placing an atomic section around the compare() function at lines Q12 and Q14. The call to the caller's compare() function is the only operation in the block, and the caller commits to making this a short function when it activates the atomic block. Therefore, the block is known to be short.

All of the other state changes in the implementation occur in local variables. Since the local variables are abandoned when qsort() returns, they pose no risk to the system's consistency when qsort() is interrupted.

An example of a call with atomic sections marked is given in gsh_qsort() atomic_sections_are_marked() and directives. These directives are shown at line Q42 and Q44 in Listing 3.7.

3.8 Case studies

We evaluated our directives by applying them to three examples taken from the PARSEC benchmark suite: BLACKSCHOLES, SWAPTIONS, and X264. The benchmarks are relatively self-contained, relying only on the code in the benchmark and the C runtime library. We applied directives to all of the shared heap state in the benchmarks and created a tollgate for the C runtime library.

[6] Technically, this is not true: both the element size and also the number of elements are bounded by the range of size_t. However, the 4GB bound placed on these parameters by 32-bit systems is effectively unbounded given the current state of hardware and the timing requirements for human-system interaction.

[7] In practice this is almost never the case. The interface to qsort() specifies that the compare() function is not allowed to modify elements of the array being sorted. In addition, the implementer of the compare() function cannot rely on the calling patterns of the qsort() routine. Therefore, the function would need to gain access to the system's state via thread-local storage or global variables. We deal with this case here for completeness.

Policies employed		
Serial thread confinement: whole blocks	2	0.4%
Serial thread confinement: sliceable arrays	1	0.2%
Modeling directives		
Whole block transfers	8	1.6%
Array slice transfers	4	0.8%
Publishes of immutable blocks	2	0.4%
Overhead directives		
Lines changed for sliceable arrays	3	0.6%
Support code	17	3.5%
Totals		
Modified lines	37	7.6%
Unmodified lines	453	92.5%

Table 3.4: Summary of changes to BLACKSCHOLES

BLACKSCHOLES

BLACKSCHOLES computes the prices of a portfolio of European options by numerically computing a partial differential equation. A master thread distributes the portfolio to a series of worker threads which work independently.

Since the threads are mostly independent, the annotation effort for the application is relatively light, as can be seen in Table 3.4. The benchmark provides a read-only array describing options. Worker threads are assigned a range of options to price when the system is initialized, and write their results into a slice of a results array after computing the prices for their options. Most of the lines of support code are for debugging output that we added to ensure that the benchmark was working. We made one non-annotation change to BLACKSCHOLES to eliminate an extra string copy of the option type. This change had a negligible effect on the performance numbers.

SWAPTIONS

SWAPTIONS runs a Monte Carlo simulation to compute the price of a portfolio of SWAPTIONS. As in BLACKSCHOLES, a master thread splits the portfolio into segments and then starts long-running worker threads to parallelize the computation.

Table 3.5 shows that SWAPTIONS makes use of serial thread confinement. We instrumented SWAPTIONS before we developed sliceable arrays. The directives in SWAPTIONS manually create a sliceable array by padding the elements of an array to page boundaries, allocating an array, and then using directives to move each index of the array into its own transferable region.

Unlike BLACKSCHOLES, the worker threads in SWAPTIONS create and free a large number of thread confined data structures. Our current runtime system does not automatically support recycling memory, and so our directives added support for recycling these data structures while maintaining thread-confinement. We discuss the performance implications of these changes in more detail in section Section 5.2.7.

X264

X264 is a lossy video encoder that is capable of processing multiple frames at one time. A master thread spawns a new worker thread for each frame, giving it references to the uncompressed data for its frame and references to prior frames, some of which are still being encoded by other workers. The workers employ a

Policies employed		
Serial thread confinement: whole blocks	1	0.1%
Modeling directives		
Whole block transfers	4	0.2%
Conversion to sliceable array (manual)	1	0.1%
Support code	91	5.0%
Totals		
Modified lines	97	5.4%
Unmodified lines	1714	94.6%

Table 3.5: Summary of changes to SWAPTIONS

combination of striping and pipelining by periodically broadcasting the number of scanlines that they have encoded. Workers encoding later frames then read these scanlines. This approach allows the workers to encode frames in parallel, relying on information about the encoding of former frames while also avoiding data races.

When we attempted to annotate the version of X264 included in PARSEC (version r1047), we found a place where the master thread read from the region transferred to the worker after the worker was started. The worker writes to this part of its region after it has started. The combination of command line parameters used in the PARSEC benchmarks does not trigger this race. Therefore it would not be detected by dynamic checkers. However, this means that developers could not consistently annotate this code. We noticed that the offending code had been eliminated in a later version (r1185) of X264, so we rolled forward to this version and completed our directives.

Modeling the sharing rules of X264 required extensive directives, as shown in Table 3.6. Much of the complexity of the model is a direct consequence of the complexity of X264's data structures: frames are represented by 8 different shared arrays. Some of these arrays directly represent scanlines and pixel values. Others represent macroblocks, a square group of 256 adjacent pixels.

The support code is largely confined to four locations in X264. First, new frames are typically allocated by the master thread as it initializes the worker thread. Therefore, the master thread must release the read-write access that it acquires to the frame before transferring it to the worker. Second, the worker threads must acquire permissions to their frames before they read and write to them and release permissions before terminating. We handle this by inserting a special wrapper around the code that implements the worker thread.

Third, frame writers must release write permissions to slices of the 8 shared buffers that represent the frame before updating the number of scanlines completed. We added this code to the routine that updates the number of scanlines completed. This code is complicated by two factors. The ranges are different for different buffers, in part because of the difference in data types in the buffers. In addition, some of the buffers are shared on a page-by-page basis to allow references to the buffer to be passed to assembly language routines. However, multiple scanlines frequently share the same page. If some of the completed scanlines share pages with incomplete scanlines we retain write access to the page and round the number of scanlines completed down to ensure that readers do not attempt to acquire the page. This code would have been unnecessary if the developers of X264 had created individual blocks for scanlines.

Finally, we added code to the routines that readers use to check the progress of shared buffers. When a reader observes the number of scanlines completed we automatically obtain read access to the relevant pages in the frame. The logic in this code closely follows the logic used by writers.

Policies employed		
Thread-local	2	0.00%
Serial thread confinement: whole blocks	8	0.01%
Serial thread confinement: sliceable arrays	8	0.00%
Shared: immutable	2	0.00%
Shared: thread safe	2	0.00%
Guarded	1	0.00%
Modeling directives		
Whole block transfers	55	0.10%
Array slice transfers	24	0.04%
Publishes of immutable blocks	2	0.00%
Overhead directives		
Splitting multi-policy blocks	59	0.11%
Lines changed for sliceable arrays	111	0.21%
Support code	444	0.82%
Totals		
Modified lines	718	1.33%
Unmodified lines	53,144	98.67%

Table 3.6: Summary of changes to x264

Other PARSEC benchmarks examined

We evaluated several other benchmarks as part of this work. CANNEAL is designed to allow races and then recover from them. Since all of our policies involve avoiding data races, we would need to treat all of the shared state as shared thread-safe.

The blocks in STREAMCLUSTER use different concurrency policies for different fields. We believe that we will eventually be able to apply our directives to STREAMCLUSTER by splitting these blocks.

We also examined FLUIDANIMATE, and have been able to annotate it with our technique. However, the very large number of fine-grained locks in the system causes resource allocation problems in our dynamic checking.

These benchmarks may not be representative of many systems that we will examine for attentiveness. First, the benchmarks are transformational systems [53], accepting input, doing some computation, and generating output. By definition attentive systems are reactive, running in a continuous loop and transforming their internal state in response to messages from their clients. Therefore, threading in these benchmarks is primarily used to diving work among multiple processors. As a result, the benchmarks make extensive use of array slicing and fine-grained locks. While some parts of attentive systems may adopt similar designs, we expect that the primary use of threading in these systems will be to provide prompt responses to the user while the system is engaged in long operations. Therefore, attentive systems are more likely to rely on long-duration locking and thread-local storage. As a result, the annotation effort for attentive systems may be lower than the benchmarks described in this chapter.

3.8.1 Limitations of the directives

Global and static variables present a particular challenge to our directives, since they are allocated and initialized before the application begins to execute. Therefore, we added associate_global(), a directive that places a global variable into the region bound to the thread executing the directive. In some cases, such as assigning a policy to static variables within a function, it may not be possible for developers to ensure that the associate_global() happens a single time. We permit multiple calls for a single global or static

variable if the policy is identical for all of the calls. By default we assume that global variables are all shared thread-safe. This is inconsistent with our default for blocks on the heap. In the future, we would like to make the default for global variables thread-local, assigning them to the initial thread.

We are also unable to assign policies to blocks allocated on thread stacks. Some threaded applications, including the BLACKSCHOLES benchmark, use allocations on the main thread's stack to allocate mutexes, condition variables, thread descriptors, and the application-defined data that is passed to new threads. In addition, the pthreads library also does some of the initialization of the child's stack from the parent thread before starting the child.

Two factors make it difficult to apply policies to blocks that are allocated on the stack. First, the compiler automatically assigns these blocks an address based on the current stack pointer. These addresses are rarely page-aligned, preventing us from using techniques that reduce the cost of checking the directives, as described in Section 5.2. Second, the blocks will automatically be deallocated if the parent thread returns from the function that originally created the block. There are several possible solutions to this problem. Our current recommendation is to move all shared blocks into the heap, managing their lifetime with malloc() and free() calls. In the future we may be able to use C++ destructors to detect cases where shared blocks are deallocated.

3.9 Conclusion

In this chapter we have introduced directives, an approach to connecting the design of an attentive system to its concrete implementation. We have described three classes of directives, one focused on consistency, one focused on promptness, and a third focused on the relationship between consistency and promptness in attentive systems. We have also introduced tollgates, a way for developers to maintain many of the benefits of directives in systems constructed by combining third-party components. Finally, we have described case studies that evaluate the directives focused on consistency by applying them to a series of third-party concurrency benchmarks.

In Chapter 4 we apply these directives when reasoning about the design and implementation of systems.

In Chapter 5 we will describe our current, partial implementations of runtime systems that support trusted and checked execution. The runtime that supports trusted execution attempts to avoid consistency failures by rolling back the entire state of the system. Therefore, it does not support the directives that describe regions and consistency. Our experience with this runtime system motivated the finer-grained approach to managing consistency described in this chapter. The current implementation of checked execution uses the directives that we have described for regions, policies, and tollgates. It does not yet support redirection, and therefore does not implement the directives for requests, dependencies, promptness, and atomic sections.

Chapter 4

Design for attentiveness

The design of an attentive system should confine responsibility for attentiveness to a small number of components. Confining responsibility for attentiveness both facilitates the reuse of components, especially components that do not directly support attentiveness, and also allows developers to focus their investigation of attentiveness failures. It is difficult to address attentiveness concerns within individual components. First, the constraints on promptness are often specific to a particular system, and the effect of any given component on promptness is determined by the way that it is integrated into the system. Second, maintaining inter-component consistency requires collaboration among multiple components. Attempting to support this collaboration within a component would create inter-component dependencies that would limit opportunities for reuse. Unfortunately, the designs of many systems delegate, either intentionally or unintentionally, the responsibility for maintaining promptness and consistency to their components. Developers using these designs cannot improve the attentiveness of these systems while avoiding changes to components.

In this chapter we consider the relationship between attentiveness and design in three stages. First, we analyze five systems and one network protocol to illustrate the connection between design and attentiveness. Next we describe two attempts to retrofit attentiveness into one of the systems by modifying the GTK+ toolkit, one of the components in the system. These attempts point to the need for system-level design to support attentiveness. Then, we outline a system-level design for attentive systems that addresses the problems that we found in the previous systems. This design assumes the presence of runtime support, which we discuss in more detail in Chapter 5. We conclude by applying the design to a simple client-server system and demonstrating that the design addresses the attentiveness failures observed in the system.

4.1 Assessment of designs

We chose to examine four systems that cover different parts of the design space for attentive systems:

- Inkscape [60]: a toolkit-based vector graphics editor
- A simple client-server system written with Java Remote Method Invocation (RMI) [96]
- Thunderbird [99]: a toolkit-based email client that uses the IMAP protocol
- RoundCube [85]: a web-based email client that uses the IMAP protocol

Inkscape is representative of a group of systems that must restrict the use of threading due to the presence of non-thread-safe code. The RMI client-server system allows us both to explore attentiveness in a system that does not involve direct human-computer interaction and also to consider the effect of communication protocols on attentive systems. Thunderbird is highly threaded, allowing us to explore attentiveness failures

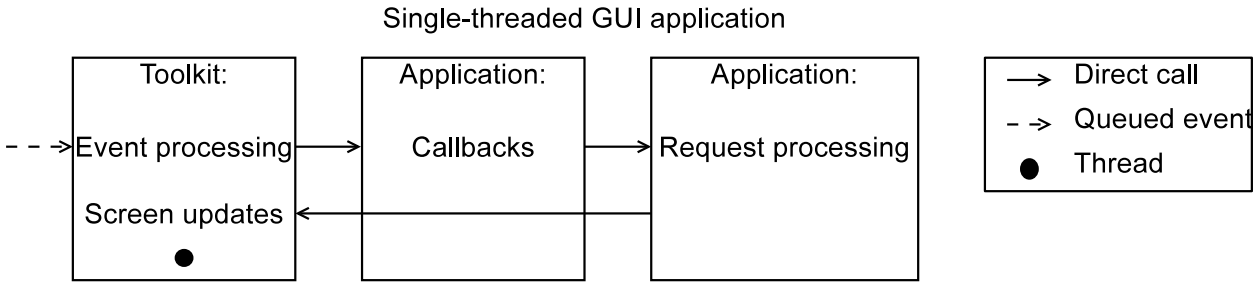


Figure 4.1: Diagram showing the structure of Inkscape from the perspective of threading. This diagram shows the system in an idle state, with the thread executing in the toolkit. Since there is only one thread, attentiveness failures occur if any request's processing time exceeds the acknowledgment time.

that occur in threaded systems. Finally, RoundCube provides similar functionality to Thunderbird but uses web technologies that isolate the user interface, which runs in the web browser, from most of the processing, which runs on the web server. This separation allows RoundCube to overcome many of the problems associated with Thunderbird. However, RoundCube continues to suffer from attentiveness failures.

4.1.1 Inkscape

The design of Inkscape is partially dictated by GTK+ [47], the GUI toolkit that provides Inkscape's user interface. GTK+ imposes restrictions on the control flow of applications that use it, requiring the application to implement the standard interaction cycle [63]. First, the application registers one or more callbacks with the toolkit. Next, the application begins the standard interaction cycle by calling gtk_main() to initialize GTK+. The function does not return until the application terminates, allowing the toolkit to use the thread that initialized the toolkit to process events. In the discussion below we call this "the toolkit thread." While the application is running the toolkit invokes one or more of the callbacks registered by the application to inform it of new requests, a property called inversion of control [62].

During the standard interaction cycle the application must not make uncoordinated calls to GTK+ from other threads. The application coordinates calls with the toolkit thread either by calling gdk_threads_enter() and gdk_threads_leave()[1] or by making the calls from callbacks, which are always executed by the toolkit thread.

GTK+ imposes these restrictions on calls because it relies on thread-local regions to protect its internal state. Calls from other threads would access the toolkit's internal state without coordination, leading to data races. This design, with minor variations,[2] is typical of most general purpose GUI toolkits. While some special-purpose toolkits are able to allow calls from multiple threads [37], prior attempts to create general-purpose multi-threaded GUI toolkits has resulted in designs that lead to errors. Either the designs force developers using the toolkit to follow complex rules for reentrant calls, or the implemented toolkits exhibit concurrency failures such as deadlocks [40] and data races [52].

The application-specific code in Inkscape places further restrictions on threading. The implementation of Inkscape predates the widespread use of threading in GUI applications. Therefore, much of the code in Inkscape is not thread-safe, in essence relying on thread-local regions to avoid races. As a result, any code that uses multiple threads must hide these threads from both the GTK+ toolkit and the non-thread-safe

[1] Technically, these calls are in the GDK library, one of the support libraries used in the construction of the toolkit. For simplicity in this work we consider GTK+'s supporting libraries to be a part of GTK+.

[2] For example, in Swing the toolkit creates a new thread to be the toolkit thread and returns from the initialization call. Applications invoke Swing methods from other threads via the invokeLater() call. In part the differences in design reflect different assumptions about the platform. The design of GTK+ assumes that threading may not be present on the platform, while Swing can assume the presence of threading.

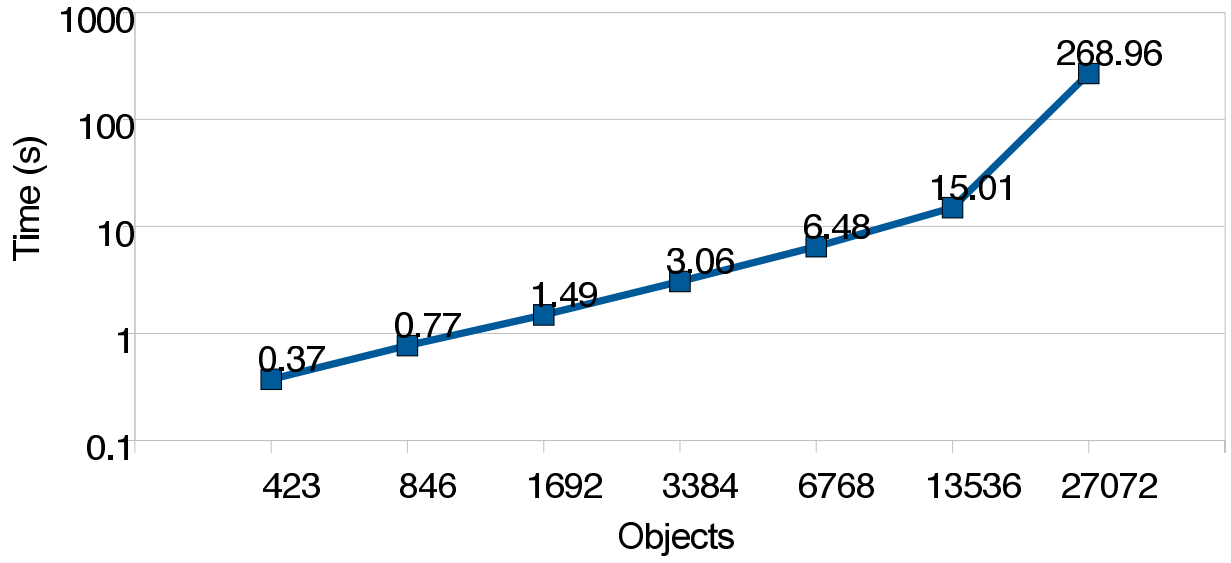


Figure 4.2: This log-log graph shows the time to complete *Paste* requests in Inkscape. For clarity the number of objects for each point is shown on the X axis, and the exact time for the paste is shown with each data point. The time varies linearly with the number of objects pasted for most of the tests, but increases with the final test due to swapping (physical memory becomes an over-subscribed resource). Even at 423 objects the time to complete the *Paste* poses a risk to attentiveness, since the acknowledgment time is roughly 0.1s.

code in Inkscape.

The resulting architecture is shown in Figure 4.1. Since there is only a single thread, attentiveness failures occur in Inkscape if the processing time of any request exceeds the acknowledgment time of the system, which is usually 100ms. This problem is widely recognized [71], and is typically avoided by employing the wrapping pattern in the callbacks to isolate the long operations from the toolkit. Since the use of the wrapping pattern is so common, it may seem to the reader that the pattern should actually be incorporated in the GUI toolkit itself. Rather than calling the callback directly, the toolkit thread would spawn a worker thread, instruct it to activate the callback, and then continue to look for requests from the user. Toolkit-based applications do not interact with their users in terms of requests. Instead, they receive low-level events and infer requests from the events, following the model used in the Xerox Alto [7]. This approach is not feasible because callbacks frequently make calls that reconfigure the user interface [74], potentially changing the interpretation of future requests. Therefore, the callback must block the toolkit thread until it has completed its reconfigurations of the user interface. In addition, the callbacks frequently make calls to the toolkit to update the view after long operations complete.

Unfortunately, the processing time of requests in Inkscape is unbounded: the execution time of many requests depends on the complexity of the document being edited. For example, Figure 4.2 demonstrates that the processing time for *Paste* operations varies, more or less linearly, with the number of objects being pasted. Therefore, a user attempting to merge two documents may cause an attentiveness failure by:

1. Opening the first document
2. Issuing a *SelectAll* request
3. Issuing a *Copy* request
4. Opening the second document

5. Issuing a *Paste* request by using the "Edit" menu

In this case, Inkscape's only thread is trapped in request processing. Therefore, if the number of objects being pasted is large, Inkscape does not acknowledge the *Paste* request by closing the Edit menu and does not provide the user with a way to submit new requests. As a result, the user has no way to redirect Inkscape.

Table 4.1 shows how an attentive system would respond to this situation. This table shows the relationship among:

- A user's goals
- His actions
- The events that arrive at the toolkit
- The toolkit's responses
- The callbacks registered by Inkscape

For example, to initiate a *Paste* request, the user clicks on a menu (A1.1), causing a ButtonPress event (E1.1) to arrive at the toolkit. The toolkit responds by opening the menu (R1.1). Since this interaction does not form a completed application-level request, no callbacks are invoked. A1.2 is also handled by the toolkit without involving Inkscape. However, A1.3 forms a completed request, causing the toolkit to call the Inkscape's paste routine at CB1.3.1. An attentive system would acknowledge the request (R1.3.2 and R1.3.3), and then listen for the arrival of a possible redirection request such as the one shown at G2.

However, Inkscape blocks before R1.3.2, failing to acknowledge the paste request until it completes. Not only does this represent a failure to respond to the request within its acknowledgment time, it also makes it impossible for the user to issue a cancel command to stop the paste operation. Given the restrictions placed on threading by GTK+ and Inkscape, developers typically use one of three solutions to address this problem:

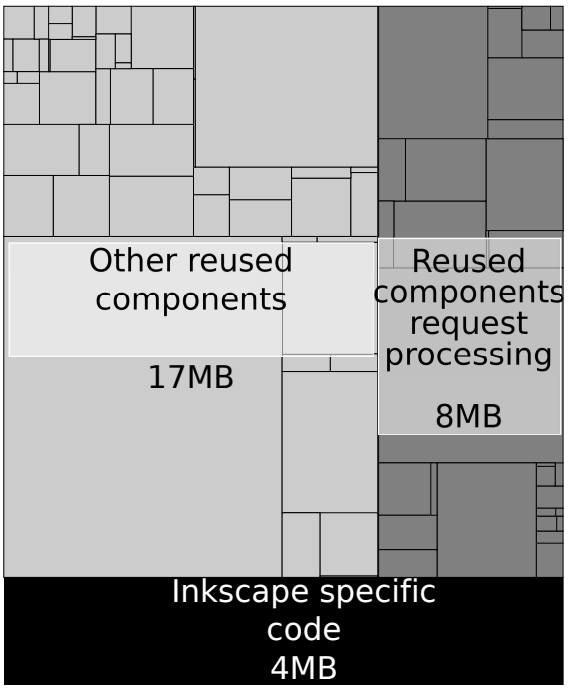
- Polling: inserting code into the components responsible for *Paste* and any other long operations to check for incoming requests and redirection. For example, the Macintosh operating system required developers to poll by calling WaitNextEvent().
- Wrapping: developing a system service that is integrated with the toolkit, supporting CANCEL without changes and cooperation from the application. We provide examples of wrapping in Chapter 5.
- Threading: refactoring the application to use multiple threads, splitting its state into two thread-local regions. For example, the Thunderbird email client uses threading.

Polling was used to support both redirection and multitasking before threads were available [5]. Even with threads, developers often find that they must use polling to stop threads without corrupting the application's state [70]. There are two disadvantages to polling. First, developers must ensure that the application polls frequently, in effect ensuring that the delay between any two poll calls in their application is short. If this is not the case, clients continue to encounter promptness failures when interacting with the application. However, developers must also ensure that they do not poll too frequently, since this both represents wasted software development effort and also can lead to performance problems [36]. Second, implementing polling can force developers to make application-specific changes to modules that they wanted to reuse. It is difficult to generalize these changes, since the definition of short is often application-specific. In addition, the length of operations within the modules is often affected by details of the application's implementation. Finally, these modifications create new dependencies between the modules that can make them harder to reuse in future applications. Applications often reuse a large number of modules. For example, Figure 4.3 shows that Inkscape, a relatively simple application, reuses 53 modules. These modules are controlled by at least 20 different open-source development teams.

Our early attempts to add redirection to Inkscape used wrapping. A wrapped system uses toolkit modifications to detect requests and assigns responsibility for redirecting requests to a separate process

Event	User		Measurement of promptness	System		
				Toolkit		Callback
	Goal	Action		Event	Response	
G1	Paste the contents of the clipboard					
A1.1		Click on the edit menu				
E1.1			Ack. time of A1.1	ButtonPress, b=0, x=75, y=10		
R1.1					Display edit menu	
A1.2		Drag down to the paste menu item				
E1.2			Ack. time of A1.2	MouseDrag x=75, y=110		
R1.2					Highlight paste item	
A1.3		Release the mouse button				
E1.3			Ack. time of A1.3	ButtonRelease, b=0 x=75, y=100		
CB1.3.1					New request: Paste	
						Processes paste
CC1.3.1						Returns to toolkit
R1.3.2					Hide edit menu	
R1.3.3					Display a cancel button	
G2	Cancel the paste operation					
A2.1		Click the cancel button				
E2.1			Ack. time of A2.1	ButtonPress, b=0, x=75, y=10		
E2.2				ButtonRelease, b=0, x=75, y=100		
CB2.2.1					New request: Cancel(C1.3.1)	
						Processes cancel
CC2.2.1			Latency of A2.1			Returns to toolkit
R2.2.2					Flash the cancel button	
R2.2.3					Close cancel dialog	
R2.2.4						C1.3.1 canceled

Table 4.1: This table relates the user's goals and actions, shown on the left, to measures of promptness, shown in the middle, to the control-flow in the system, shown on the right. The user-system boundary is illustrated by the double line, and the toolkit-application boundary is shown by the dashed line. The table illustrates one of the flaws frequently encountered in single-threaded designs: the entire *Paste* operation, shown on line CB1.3.1, is a part of the acknowledgment time of the request.



Other reused

Package	NCNB LOC	%
libkdefx	568934	14.59
libqt-mt	568415	14.57
libxml2	195448	5.01
libstdc++	119235	3.06
libpangomm-1	91622	2.35
libgnutls	83468	2.14
libgnomevfs-2	72784	1.87
libdbus-1	62489	1.60
libaudio	48086	1.23
libORBitCosNaming-2	38446	0.99
libgconf-2	31951	0.82
libavahi-glib	31349	0.80
libXt	31314	0.80
libxslt	30122	0.77
libgcrypt	25503	0.65
libgc	21573	0.55
libbonobo-activation	20468	0.52
liblcms	20016	0.51
libjpeg	18696	0.48
libglibmm-2	12807	0.33
...		
Total	2152300	55.18

Request processing

Package	NCNB LOC	%
libutil-2	576009	14.77
libgtk-x11-2	416547	10.68
libgthread-2	124575	3.19
libX11	100008	2.56
libfreetype	77043	1.98
libcairo	56946	1.46
pango-basic-fc	55916	1.43
libpng12	29294	0.75
libfontconfig	21271	0.55
libz	15547	0.40
libexpat	12982	0.33
libatk-1	7256	0.19
libXext	3837	0.10
libXcursor	2809	0.07
libXrender	2738	0.07
libXi	2612	0.07
libXdmcp	1116	0.03
libXfixes	996	0.03
libXrandr	654	0.02
libXau	582	0.01
libXinerama	228	0.01
Total	1508966	38.7

Figure 4.3: Inkscape source study as a treemap. The size of each box represents the number of non-comment, non-blank lines of code in the component. The black box represents the source code for Inkscape. All of the other boxes represent third-party libraries that are invoked by the application as it executes.

called a cancellation manager. The cancellation manager relies on operating system services to redirect the wrapped application. Wrapping has disadvantages in terms of efficiency and consistency. We describe our work with wrapping in more detail in Section 4.2.1 and Section 4.2.2.

Finally, it is possible to address redirection in Inkscape by introducing an additional thread to handle request processing. If communication between these threads is carefully constrained, it is possible to do this while preserving the thread-local regions of both GTK+ and Inkscape. These constraints informed the general pattern in Section 4.3. Runtime support, described in Chapter 5, ensures that developers do not access thread-local regions from the wrong thread.

4.1.2 RMI client and server

Client-server systems can also suffer from attentiveness failures. Some attentiveness failures are caused by factors beyond the control of the system's designers, such as network failures. Others can be addressed by careful design of the client, server, and their communication protocol. In this section we examine a failure that is present in many client-server systems that can be addressed by design. We use a simple client-server system written in Java to produce this failure. The client and server in our example communicate via Java Remote Method Invocation [96] (RMI), a library that allows developers to communicate with remote objects using method calls on local proxy objects.

In our example, clients submit requests to a server for remote execution by encapsulating them in objects that implement the IRequest interface. This interface contains a single method, execute(), that has a generic return type. The client passes the request objects to the executeRequest() method provided by the server. RMI serializes the client's request object and transmits it to the server, which deserializes the request object, calls the request's execute() method, and returns the method's result to the client.

There are two potential attentiveness failures in the example, one affecting the client and the other affecting the server. The client will not be attentive if it encounters a slow server. Since RMI mimics standard method calls, which do not provide the ability to cancel calls in progress, the client has no way to regain control of threads blocked by calls to slow servers.

Developers have several options for working around attentiveness failures on the client. First, they could use a separate thread to issue RMI calls, allowing the client to resume operation before receiving the result of the call. Second, they can use extensions to RMI, such as Interruptible RMI [77]. These extensions allow a client to cancel blocked RMI calls by closing the socket used to communicate with the server.

The implementation of interruptible RMI relies on the implementation details of sockets in Java. Specifically, the sockets must permit close() to proceed while threads are blocked in other methods of the socket. To provide this functionality developers must be willing to accept data races in certain classes [44], such as BufferedInputStream.[3] The implementation in JDK 1.2 added synchronization in the send(), receive(), and close() methods to resolve some race conditions that could occur when close() was called while another thread was accessing the object. The new synchronization caused calls to close() to block, breaking code that relied on close() to interrupt the send() and receive() methods. Eventually, the designers removed the added synchronization, reintroducing the race conditions but allowing developers to use close() to regain control of threads blocked in send() and receive().

Attentiveness failures also occur on the server. An attentiveness failure occurs when clients abandon requests, either by closing sockets while the request is being processed or by crashing. From the server's perspective, an abandoned request can also occur if a long-term communication failure occurs while the request is being processed. This scenario can be observed on high-traffic web sites, since browsers make it particularly easy for users to abandon requests by pressing the refresh button. The web browser responds to the refresh button by dropping the current HTTP connection to the server and opening a new connection to re-send the request. When this behavior occurs at scale, it produces a troublesome form of positive

[3] We are indebted to Aaron Greenhouse for pointing out this example.

feedback: the site experiences a significant increase in request submissions as it becomes overloaded. Some communication libraries, such as Interruptible RMI, allow servers to poll for abandoned requests. However, these libraries often must be used on both the client and the server to be effective, and the polling spreads the responsibility for detecting abandoned requests throughout the server's implementation.

In our simple system, the effect of abandoned requests is easy to observe. A request to calculate the first 4,500 digits of π takes 2s. If the client submits and then abandons a request to calculate 450,000 digits of π, the time to complete the 4,500 digit calculation increases to 4.4s.

Our design, descried in Section 4.3, addresses both of the problems described above. Servers can address and terminate abandoned requests promptly, even in the presence of communication failures. In addition, the design assigns responsibility for detecting abandoned requests to a small part of the server that is independent of the reused components.

4.1.3 Thunderbird

Applications that employ multiple threads are also subject to attentiveness failures. Thunderbird 2.0 [99], a mail user agent, makes extensive use of threads but exhibits attentiveness failures when interacting with an IMAP [23] server. The IMAP server maintains a copy of each of the user's messages, grouped into various folders. Thunderbird and other IMAP clients synchronize with the folders on the IMAP server, both updating their local copy of the messages at startup and also updating the IMAP server as users move, copy, and delete messages. Frequent synchronization ensures that users interacting with multiple computers see the same set of messages on each computer.

Thunderbird minimizes the user intervention required to synchronize with the IMAP server, initiating synchronization as a background task when a user opens a folder. Meanwhile, Thunderbird displays a message list using the contents of its cache. The synchronization task communicates with the IMAP server, checking for changed messages, and possibly downloading some messages for the junk mail scanner. Most of this activity is invisible to the user.

In its implementation, Thunderbird uses the Active Object pattern [69], which places most objects into thread-local regions. Other threads manipulate the objects by sending asynchronous messages to the thread associated with the object's region. As a result, the Active Object pattern greatly reduces the risk of data races; while the asynchrony in the messaging system reduces the chance that blocking will result in promptness failures.

However, we have discovered that Thunderbird suffers from a variety of promptness failures. These failures arise because Thunderbird fails to prioritize the assignment of network bandwidth, a constrained resource, to the user's requests. When bandwidth is over-allocated, Thunderbird promptly acknowledges requests from its user to redirect but fails to complete the redirection. The following scenario illustrates the problem:

1. The user switches to Folder A
2. Thunderbird opens Folder A on the IMAP server, detects new messages, and begins to download their headers
3. The user switches to Folder B
4. Thunderbird opens Folder B on the IMAP server, detects new messages, and begins to download their headers
5. The user switches back to Folder A
6. The user clicks on message A1 in the thread pane, issuing a *ViewMessage* request
7. *One minute later* Thunderbird displays message A1
8. The user clicks on message A2 in the thread pane, issuing a *ViewMessage* request

9. *Five minutes later* Thunderbird still has not downloaded the message

Unlike Inkscape, Thunderbird acknowledges the *ViewMessage* request well within the 100ms response time by highlighting the selected message in the message list. However, the message pane either remains blank or continues to display the previously selected message.

The promptness failure between steps 8 and 9 occurs because Thunderbird allows the junk mail scanner to consume most of the available network bandwidth rather than allocating bandwidth to the user's requests. When the user opens folders A and B, the junk mail scanner discovers many new messages. It begins to download these messages to scan them, quickly creating a queue of requests that consumes the bandwidth to the IMAP server. During the scenario described above, Thunderbird downloaded 720 messages between steps 8 and 9, creating the five minute delay. To make matters worse, Thunderbird occasionally wastes network bandwidth by downloading the same message twice, even using two different IMAP connections at the same time for the downloads. These problems disappear when we deactivate the junk mail scanner.

Thunderbird also suffers from multiple consistency failures. For example, Thunderbird can send spurious responses during the delay described above. This can happen when the user selects a second message while waiting for a message to download from the server. Thunderbird acknowledges the request by highlighting the second message in the message list. The user's expectation is that the next update of the message pane will show the selected message. However, Thunderbird may continue to download the first message, displaying it in the message pane when the download completes. The user may be confused by this update, either associating the content of the first message with the sender of the second message or concluding that there is a permanent loss of consistency in Thunderbird's internal state.

Finally, Thunderbird and the user can disagree about the interpretation of a request. This happens when Thunderbird detects that a message has been deleted from a folder during the background synchronization task. Thunderbird may then repaint the message list, shifting the position of messages to remove the deleted message. As a result, the position of a message just as the user is clicking on it. As a result, Thunderbird displays a different message than the one that the user intended.

Our analysis of Thunderbird informs several parts of our design:

- Attentive systems must manage all scarce resources, not just processor time and threads. The failure in Thunderbird occurs because its design addresses only blocking in threads as a risk to attentiveness.
- A single component in the system should be aware of all of the system's activities. This allows the system to inform the user of its current activities and also allows the system to detect and eliminate redundant work.
- Activities should have priorities. Requests coming directly from users should generally have higher priority than background tasks.
- The priority of an activity may change. For example, the junk mail scanner originally scheduled a download of A2. However, the priority of downloading A2 increased when the user issued a *ViewMessage* request.

4.1.4 IMAP

The design of the IMAP protocol contributes to the attentiveness failures that we observed in Thunderbird. Some of these failures originate in IMAP's data model. Each account is associated with one or more folders. The number of folders is not bounded. The number of messages in a folder is theoretically bounded: each message in the folder must be referenced by a unique unsigned 32-bit integer. The size of each message is also theoretically bounded: the size of the message in octets must be representable by a 32-bit integer. Since both of these bounds are large relative the available network bandwidth, we choose to design systems as if they do not place bounds on the length of operations.

Messages also contribute to attentiveness problems. While the content of a message is immutable, applications can delete the message, copy it to another folder, and attach and remove flags. For example, many IMAP applications use flags to mark messages that have been scanned for junk mail. In addition, many applications expose the ability to set flags to their users, allowing them to prioritize messages.

IMAP applications assume that their users will manipulate the messages stored on the server from multiple systems. Therefore, when an IMAP application is started it checks for updates on the server. Ideally, from the application's point of view, it would be able to do this by retrieving a log of updates that occurred after it was disconnected.

However, IMAP servers do not maintain change logs for folders. Instead the application must open each folder, check for new messages, check for deleted messages, and retrieve the flags of every message that remains in the folder. Since the bounds on the number of messages are relatively high, IMAP applications must assume that these are long operations. If these operations are allowed to block user requests the application will exhibit attentiveness failures.

However, treating these operations as low priority background tasks creates a risk of consistency failures. When applications display the contents of the folder in a list, these checks can cause the position of individual messages to shift as the application discovers new and deleted messages. An application accepting a *ViewMessage* request may discover that the message was deleted from the IMAP server when it attempts to retrieve it. In addition, an unexpected shift in messages may cause the user to click on the wrong message in the list.

The large bound on the size of IMAP messages also presents problems when applications attempt to download messages. The IMAP protocol does not permit redirection. Therefore an application downloading a large message is left with two choices. First, the application could download the entire message, in effect losing access to the IMAP connection in question until the download completes. This approach maximizes the efficiency of the message transfer, and does allow some forms of redirection. The application can CANCEL the download by closing the IMAP connection before the download completes. The application can also PAUSE the download by not reading bytes from the socket associated with the IMAP connection. This will cause the socket's TCP receive window to close, forcing the server to stop sending packets. Unfortunately, the server has no way to know that this behavior is due to redirection, and may interpret it as evidence of either a network failure or a malfunctioning IMAP application.

Second, the application may attempt to download the message in segments. The IMAP protocol directly supports this functionality. However, it leaves the problem of determining the correct segment size to the application. If the application chooses a small segment size it can redirect quickly, but message downloading is inefficient due to the large number of small requests. If the application uses larger chunks it gains efficiency at the expense of longer redirection times. Unfortunately the optimal segment size is very dependent on the bandwidth and latency of the network connecting the application and server. Both the latency and bandwidth can change in ways that cannot be predicted by applications. For example, the user may decrease the available bandwidth by initiating a large file transfer from a different application.

The message identifiers used by the IMAP protocol lead to redundant work when an application moves a message from one folder to another. When the message arrives in the destination folder it appears as a new message and has a new identifier.[4] Therefore applications will re-download the message, even when they have a copy of the message in their cache.

Finally, the IMAP protocol does not provide redirection for some commands that are very likely to be long. For example, applications are able to initiate full-text searches across all of the messages stored in a folder. These searches may take many minutes to complete, and users often decide to change the search criteria while they are waiting. Unfortunately, the redirection approaches adopted for message downloads—

[4] In fact, in standard IMAP this is also true for the application moving the message. However, IMAP servers that support the UIDPLUS extension provide the new unique identifier for the message to the application initiating the move. Unfortunately many IMAP servers do not support the UIDPLUS extension.

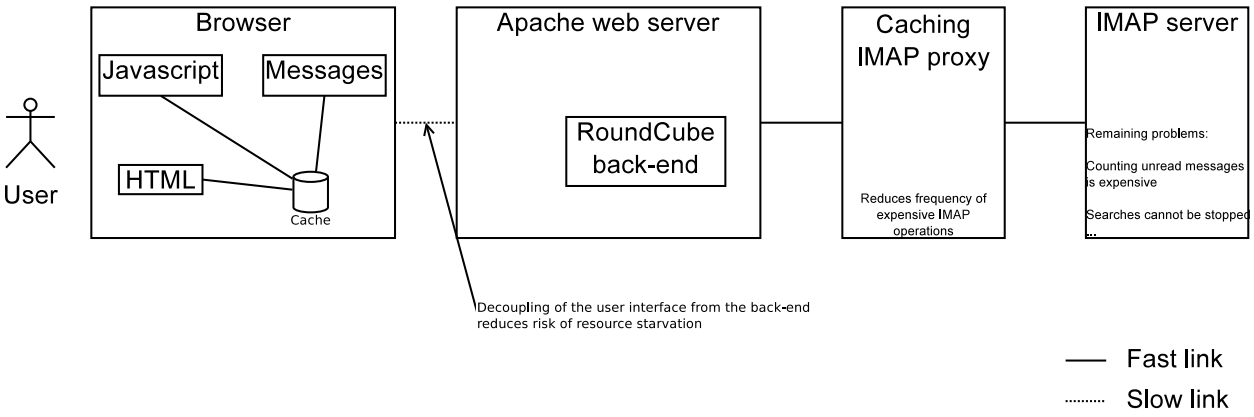


Figure 4.4: RoundCube, an AJAX email application, creates a clear separation between the user interface and the back-end. As a result, it avoids many of the attentiveness failures of Thunderbird.

closing the connection or refusing to read the results of the command—will not be effective in these cases. In most servers closing the socket will result in orphaned requests. Refusing to read the results is ineffective because the volume of results in often small, fitting within the TCP transmit window. In addition, may servers delay transmitting results until the search is completed.

Therefore, the IMAP protocol is not well suited to the construction of attentive applications. A more attentive version of the protocol would:

- Provide a log that would reduce the cost of synchronization when an application reconnects
- Allow the server to promptly detect dropped connections and CANCEL requests submitted by the connection
- Provide identifiers for messages that are independent of folders
- Allow applications to redirect requests after submitting them to the server

We believe that these recommendations would also apply to many other network protocols.

4.1.5 RoundCube

The reader may conclude that the deficiencies of the IMAP protocol enumerated above make it impossible to avoid the attentiveness failures observed in Thunderbird. However, we believe that a carefully designed IMAP application can mask many of these problems from its users. For example, the RoundCube webmail application uses the IMAP protocol to access messages, but manages to hide many of the deficiencies of IMAP from its users.

The attentiveness of RoundCube is largely a result of its design, which is shown in Figure 4.4. This design benefits from the following properties:

- Collocation of processing with data, both minimizing the impact of network delays and also minimizing the use of network bandwidth
- Caching of data to further reduce the use of network bandwidth
- Reservation of resources for the user interface, ensuring that it will remain responsive during long-running background tasks
- Limiting the use of background tasks and minimizing their size

RoundCube exploits collocation in two ways. First, RoundCube uses AJAX [88] to implement the user interface in the web browser. As a result, simple requests are handled locally, reducing both reducing their latency and also reducing their consumption of bandwidth. Second, RoundCube initiates bandwidth-intensive IMAP requests on the back-end. The back-end summarizes the requests and transmits the summaries to the user interface. As a result, RoundCube consumes network bandwidth on the back-end, where it is relatively abundant, to conserve bandwidth between the user interface and the back-end, which is often constrained.

RoundCube exploits the browser cache to further reduce its consumption of bandwidth. For example, the use of AJAX cuts the number of page refreshes dramatically when RoundCube is compared to other webmail applications. In addition, the cache is likely to retain copies of email messages, reducing the time consumed when a user views the same message multiple times.

The user interface often executes on a different machine than the back-end. As a result, the user interface and back-end have different pools of resources. Therefore the performance of the user interface is not directly affected by the resources used in back-end computations such as searches. Network bandwidth is the only shared resource, and both the front-end and back-end benefit from minimizing their use of this resource. As a result, back-end activities almost never adversely affect the performance of the user interface.

Developers have minimized the number of and extent of background tasks. There are two types of background tasks in RoundCube. First, RoundCube treats searches as background tasks, allowing users to change search parameters and switch folders while a search is in progress. This feature provides attentiveness at the user interface, but results in abandoned searches on the server similar to the abandoned requests described in Section 4.1.2.

Second, RoundCube uses a series of background tasks to collect the data needed to initialize its user interface:

1. It retrieves the list of subscribe folders to initialize the folder pane
2. It retrieves the number of unread messages in the INBOX
3. It retrieves the first 100 messages of the INBOX to initialize the thread pane
4. It retrieves the number of unread messages in all of the subscribed folders

The fourth step is very expensive and, while important, does not need to be completed before the user interface is usable. Therefore RoundCube postpones it until the first three steps are complete.

As a result of these design decisions, RoundCube generally exhibits fewer attentiveness failures than Thunderbird when working with large IMAP accounts, even when the network bandwidth between its front-end and back-end is constrained. This assessment highlights the need to carefully manage the allocation of resources in systems and also points to the need for cancellation of requests in distributed systems. We address both of these concerns in the design described in Section 4.3.

4.2 Assessment of prototypes

In the early stages of this research we developed two approaches to support attentive systems. Experience with the first system demonstrated that overhead added to the processing of short requests could compromise the promptness of the system if not carefully managed. The second system demonstrated that it is difficult to retrofit attentiveness into a system without the knowledge of the relationship between its low-level activities and the requests submitted by its users. These lessons motivated the design proposed in Section 4.3.

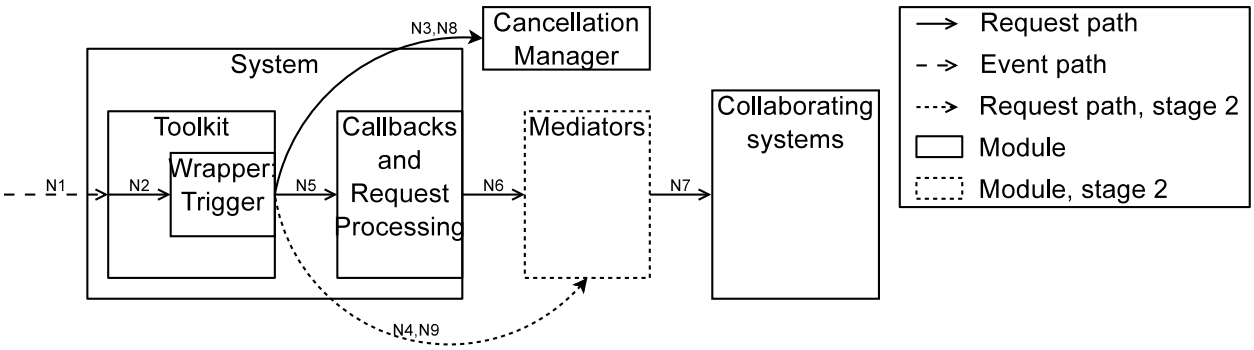


Figure 4.5: Lightweight checkpointing in Inkscape

4.2.1 Wrapping with checkpoints

In our initial attempts to improve the attentiveness of systems, we focused on adding responsibility for attentiveness through the GTK+ toolkit, one of the components in many of our examples. We called this design a wrapper, since it intercepted requests before they appeared at the application. We applied two variants of this design, described in this section and the next, to Inkscape. We assessed the wrapper against the following goals:

- Avoid modifications to the application. One consequence of this decision was that the wrapper could support only the CANCEL behavior, since other behaviors rely on application-specific knowledge of the requests
- Achieve sufficient efficiency to avoid introducing new promptness failures due to overhead introduced by the wrapper
- Avoid consistency failures, both within the application and also between the application and the X server, a collaborating system

The wrapper was implemented with four components, which are shown in Figure 4.5. First, we introduced a new component into the GTK+ library, called a trigger, to detect new requests and coordinate redirection. The trigger is implemented within the gtk_propagate_event() function, the last function in the GTK+ toolkit that encounters every request from the user. Second, we created a separate process called the cancellation manager (CM). The cancellation manager displayed and monitored a cancel button whenever Inkscape processed a request. Third, we implemented a system call in the Linux kernel to capture and restore checkpoints of Inkscape's state. The checkpointer is implemented as a new system call in the kernel, allowing it to use copy-on-write to reduce the cost making copies of the process's address space. Our implementation uses existing code in the kernel, allowing us to implement the checkpointer and restorer with about 100 lines of new code. Finally, we use a mediator to restore consistency between Inkscape and the X server when requests are canceled. This component was added in the second wrapping approach and is discussed in greater detail in the next section.

The flow of requests in the trigger is given in Figure 4.5. For completeness, we describe the interactions with both the cancellation manager and mediator below, even though the mediator was not implemented in the initial system.

Events enter the system via N1. The toolkit processes these events, generating a stream of requests. An example of this process is given in the discussion of Table 4.1 on page 66. The request stream then enters the trigger via N2. The trigger creates a checkpoint, and notifies the cancellation manager that a request is about to execute, giving it the checkpoint identifier at N3. The cancellation manager ensures that a cancel button is displayed to give the user the ability to cancel the request. The trigger then notifies the

mediators of the new request at N4. Finally, the trigger forwards the request to the application's callbacks at N5, allowing the application to begin processing the request. If the application interacts with collaborating systems, the mediators intercept the outgoing requests (N6), logging and modifying them before sending them to the collaborating systems (N7). If the request completes, control returns to the trigger, which notifies both the cancellation manager (N8) and the mediators (N9) that the request has completed. The cancellation manager runs as an independent process, allowing it to execute in parallel with the application without threatening its consistency.

The processing of CANCEL requests begins when the cancellation manager detects that the user pressed the cancel button while another request was active. The cancellation manager begins responds by using a system call to restore the application's state from the checkpoint created by the trigger. This process also affects the stack and registers of the application, causing it to re-execute the trigger code at N2. The trigger detects the rollback by examining the result of the checkpoint() system call. It then acknowledges the rollback by sending N3' to the cancellation manager and N4' to the mediators. For more details on the interaction with the mediators, see the discussion in the next section. Finally, the trigger returns control to the toolkit without invoking N5.

We implemented a prototype of our CANCEL wrapper in the Linux 2.6.14 kernel and ran it on a system with a Pentium 4 CPU running at 1.8 GHZ with 1 GB of RAM. We used a reference SVG document that contained detailed street data for Pittsburgh, Pennsylvania for our testing. This document contained approximately 4000 objects. While this document may appear to be complex, it is representative of the documents routinely edited by cartographers at companies like MapQuest. We were able to produce a ten minute attentiveness failure by copying all of the shapes in the document and then pasting the shapes into a second document of similar complexity.

Tests of the modified system confirmed that checkpointing was effective, allowing us to cancel arbitrary operations in Inkscape without noticeable delays. The wrapper was able to return control to the user within 52ms to 142ms of receiving the CANCEL request. The system call responsible for restoring the checkpoint accounted for 10ms of this time.

However, the modified system exhibited other attentiveness failures due to the overheads added by the trigger. Most of this overhead was due to the cost of creating a new checkpoint for each request, which was approximately 20ms in our system. The overhead was added to all requests, even "requests" such as tracking the mouse that occurred with high frequency and had short latencies. Even though the operations for creating and destroying checkpoints were short, our system was unable process these requests as quickly as the X server sent them. As a result, requests accumulated in Inkscape's event queue. Eventually the queue would fill, causing Inkscape to block for several seconds.

Redirecting Inkscape also caused a consistency failure when the X Server and Inkscape began to disagree about the sequence numbers for requests. The X protocol uses an implicit sequence number for requests that is rarely transmitted on the wire. Since Inkscape generated a small number of X requests after the checkpoint was created, restoring the checkpoint would cause it to repeat the sequence numbers. Eventually Inkscape would notice that the sequence numbers were misaligned and stop communicating with the X Server.

We identified the following problems with the first prototype that informed our future designs:

- Adding even a short overhead to every request can create a promptness failure
- Mediators are needed for systems that interact with X servers, even when the system does not appear to interact with the server before redirection

4.2.2 Wrapping with redo and mediators

The prototype described in this section employs three related strategies to address the problems with the first prototype. First, the prototype avoids adding checkpoints to most short operations by reusing check-

```
RC1  const time_t max_time_for_cancel = 5 * SECONDS;
RC2  bool checkpoint_exists=false;
RC3  chkptid_t checkpoint_id;
RC4  time_t cur_cancel_time = 0;

RC6  void done_with_checkpoint() {
RC7    free_checkpoint(checkpoint_id);
RC8    checkpoint_exists = false;
RC9    cur_cancel_time = 0;
RC10 }

RC12 void issue_request(Request r) {
RC13   if (!checkpoint_exists) {
RC14     checkpoint_id = create_checkpoint();
RC15     checkpoint_exists = true;
RC16   }
RC17   time_t start_time = gettime();
RC18   bool cancelled = r.execute();
RC19   time_t end_time = gettime();
RC20   if (!cancelled) {
RC21     cur_cancel_time += end_time - start_time;
RC22     if (cur_cancel_time > max_time_for_cancel) {
RC23       done_with_checkpoint();
RC24     } else {
RC25       add_redo(r, checkpoint_id);
RC26     }
RC27   } else {
RC28     provide_estimate_for_progress_feedback(cur_cancel_time);
RC29     restore_checkpoint(checkpoint_id);
RC30     while (has_redos(checkpoint_id)) {
RC31       Request rr = get_redo(checkpoint_id);
RC32       rr.execute();
RC33     }
RC34     done_with_checkpoint();
RC35   }
RC36 }
```

Listing 4.1: Checkpoint reuse expressed in pseudocode

points. Second, the prototype includes a mediator that monitors the communication with the X server, resynchronizing communication in the event of redirection. Finally, the modified wrapper must record requests completed after the checkpoint was established and replay these completed requests if the checkpoint is restored. We describe the design of these three aspects of the system below, concluding with lessons that informed our proposed design.

Checkpoint reuse

The checkpointing strategy in the first prototype created attentiveness failures because it established a new checkpoint before processing each request. Some requests such as mouse tracking requests were extremely short, very frequent, and could accumulate in queues. As a result, the checkpointing time could build up, creating promptness failures in the wrapped system.

Systems could avoid this problem by not creating a checkpoint for short requests. However, it may not be possible for the system to accurately identify short requests. In systems using this approach, if a long request were misidentified as a short request there would be no way to redirect the request after it began to execute.

Instead, we adopted an approach that identified short requests when they completed their execution. At this point the system can directly observe the execution time of the request, eliminating the risk of misidentification. In the modified system a checkpoint is in place when every request executes, ensuring that the request can be redirected. However, checkpoints are not destroyed after short requests. Therefore, in a sequence of short requests only one checkpoint is created. The checkpoint will be reused for subsequent short requests, and will only be destroyed when the estimated replay time for the sequence becomes a significant risk to the promptness of redirection requests.

The modified checkpointing strategy in the trigger follows the one shown in Figure 4.5 with some variations described below. The trigger maintains three pieces of persistent state: a current checkpoint, a timer, and a redo log of completed requests. The trigger checks to see if a checkpoint already exists before creating a new checkpoint. If no checkpoint exists trigger creates one and sends the identifier to the cancellation manager at N3.

If a checkpoint is already active, the trigger consults the timer associated with the checkpoint. The timer tracks the amount of time that was used by requests that have completed after the checkpoint was created. This timer provides an estimate of the overhead that would be added to a CANCEL the incoming request if the trigger reused the current checkpoint. If the trigger determines that this overhead is unacceptable, it destroys the current checkpoint, resets the timer, clears the log of completed requests, and creates a new checkpoint. Otherwise, the trigger proceeds through steps N3 and N4.

Before calling N5 the trigger starts the timer. When N5 returns, the trigger stops the timer[5] and adds the completed request to the redo log. The trigger does not destroy the checkpoint when the request completes, allowing future requests to reuse the checkpoint. The logic used by the trigger is shown in Listing 4.1.

In the event of a CANCEL, control returned to the trigger just after N3. After detecting that a request was canceled, the trigger would consult its queue of completed requests. The trigger then replays these requests by reproducing the calls at N5. The details of the replay are discussed in greater detail at the end of this section. Assuming that the checkpoint restoration is non-destructive, the mediator can continue to use the current checkpoint for future requests.

[5] We are assuming a single-threaded design like Inkscape, where the entire processing time for the request happens at N5. This assumption is not valid for applications that a different thread to execute the request.

Mediators

We added a mediator to the wrapper to monitor the communication between the application and the X server, a collaborating process that manages the low-level elements of the user interface such as windows and mouse events. The trigger activates the mediator at N4 in the diagram, and also initiates communication with the mediator when requests at N4 when requests are CANCELed. Communication between the application and the X server is governed by the X protocol, which attaches a sequence number to each request coming from the application. The protocol does not transmit this sequence number to conserve bandwidth. In addition, requests coming from the application are buffered and sent in batches. Therefore, it is impossible for the mediator to determine the precise relationship between the requests submitted to the application by the user and the requests generated by the application and sent to the X server. Therefore, the trigger provides the current sequence number to the mediator at N4 before starting a request.

During redirection, the application initiates communication with the mediator at N4 to inform it that the request has been canceled. Redirection is asynchronous, and therefore can occur while the application was transmitting a request to the mediator. Therefore, the application communicates with the mediator over a separate socket during redirection. Redirection follows the following steps:

1. The socket used to forward X requests from the application to the mediator is destroyed and replaced with a new socket. This eliminates the risk that a partial X request that is buffered in the kernel will confuse the mediator after redirection.
2. The mediator notifies the application of the current sequence number. This is likely to be higher than the application's current sequence number, which was restored with the checkpoint
3. The mediator then reverses the effects of any requests that were forwarded to the X server for the canceled request

We believe that mediators for other protocols would share similar properties.

Replaying requests

Given the design of our trigger, there are three options for implementing replay that correspond to different levels of abstraction in the application: at the toolkit interface, at the Xlib interface, and at the system call level. These levels are shown in Figure 4.6. We decided that redirection at the toolkit level was not likely to succeed, discovered that redirection at the Xlib level was not feasible, and obtained partial success with redirection at the system call level.

Redirection at the toolkit level is complicated by reentrant calls between the application and the toolkit. While we normally discuss the interaction between the toolkit and the application in terms of inversion of control, in reality control crosses between the toolkit and the application multiple times. For example, during a paste operation control crosses the boundary approximately 200 times, and there are reentrant calls that are nested to seven layers, as shown in Figure 4.7. Lower levels of the architecture do not feature this level of complexity, greatly simplifying the implementation replay.

Replay can be greatly simplified at the Xlib level. Xlib provides a communication channel to the X server, which provides events to the application, manages windows, and provides simple drawing commands. Replay at this level involves capturing the events that were provided to the toolkit, storing these events in a buffer that will survive checkpoint restores, and sending the buffered requests back to the toolkit.

In practice replay at the Xlib level frequently deadlocked. After some careful investigation, we discovered that the GTK+ toolkit was gaining access to the socket used to communicate with the X server. Evidence of this access is shown in Figure 4.6. At P1 in this diagram GTK+ is calling poll(), a system call that blocks until data appears on a socket. In most cases this blocking is handled within Xlib, as can be seen by the calls

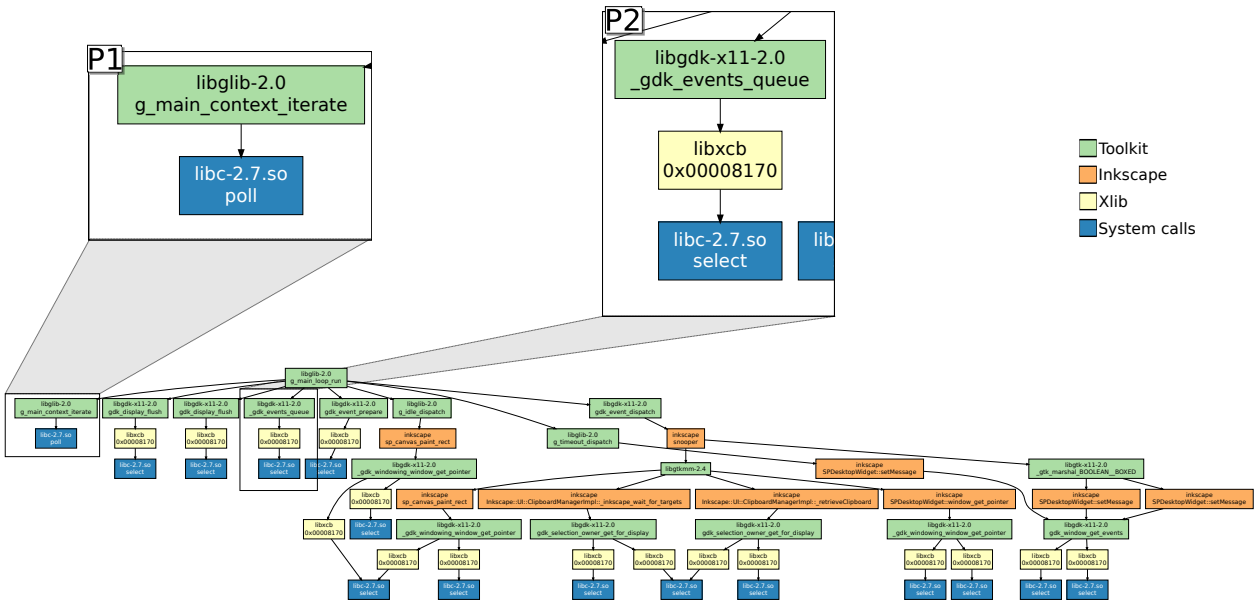


Figure 4.6: This simplified call graphs shows that GTK+ uses multiple system calls to access the socket coordinating communication with the X server. As a result, it is difficult to replay X events without a deadlock.

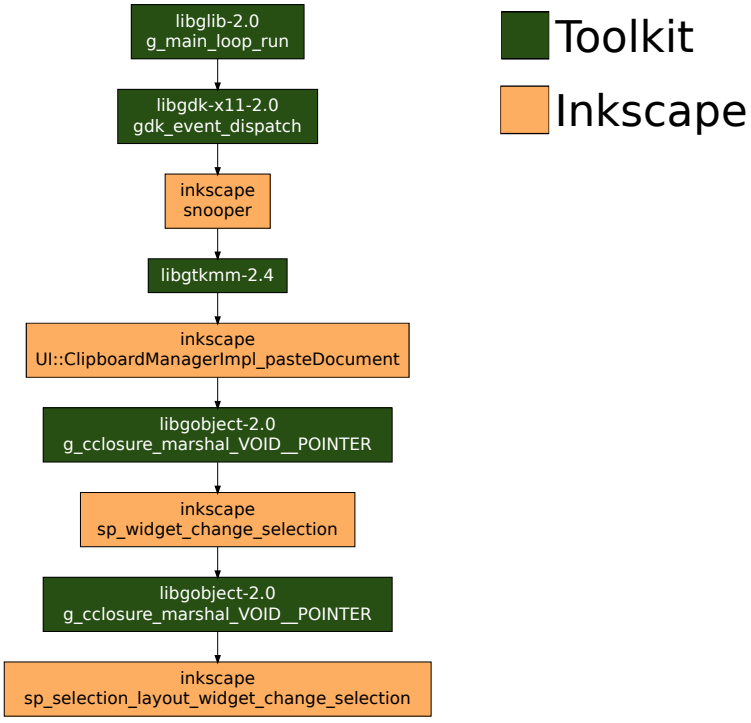


Figure 4.7: Inkscape makes a large number of reentrant calls to GTK+ (its toolkit) as it processes requests. This figure shows only the deepest set of reentrant calls made during a simple *Paste* request. The figure eliminates calls that do not cross the boundary between Inkscape and GTK+, and also eliminates the calls that are not as deeply nested. The nesting shown in this diagram makes it difficult to replay requests by simulating the interactions between the toolkit and Inkscape.

to select()[6] in Xlib, one of which is shown in P2. During replay, events are not being provided by the X server. Instead they are coming from a replay buffer. Therefore, the poll() system call will block. Unfortunately, the asynchronous nature of the X protocol prevents the mediator from coordinating the replay. Delivering events at the wrong time can cause Inkscape to misinterpret the event stream, leading it to identify the wrong requests. Therefore, we abandoned replay at the Xlib level in favor of replay at the system call level.

Our second relay approach focused on simulating the results of system calls. We used shared libraries to intercept the system calls made by Inkscape, placing these calls in a log along with the information needed to simulate their effects. When control returned to the trigger at N5 it placed a marker in the log, indicating that the calls in the log before this point were associated with the completed request. This form of replay worked for single-threaded applications. Unfortunately we determined that it would be difficult to support the same approach in a threaded application processing multiple requests in parallel. Not only would we need to maintain the logs on a per-thread basis, we would also have to examine the logs for dependencies among the requests. These dependencies arise because requests can use system calls to communicate as they execute. For example, the futex() system call implements a form of inter-request communication by managing access to contended locks. Other system calls are ambiguous. For example, two requests can communicate through a pair of connected sockets via read() and write() system calls. In other cases read() and write() system calls represent communication with collaborating systems that must be managed by mediators. Finally, the same calls may indicate that changes are being made to the filesystems that must be undone in the event of redirection. Differentiating these cases would involve a detailed study of the semantics of the calls.

Finally, all of these replay approaches encounter problems with the mode changes that occur in the toolkit. Some of the reentrant calls made by the application change the toolkit's interpretation of future events, thereby changing its detection of requests. Unlike the problem of reentrancy, it is not possible to bypass this problem at the toolkit level or below. The *Paste* scenario in Inkscape provides one example of this problem. If the user initiates the paste operation by pulling down the *Edit* menu, dragging to *Paste*, and releasing the mouse button, the toolkit will forward the request to Inkscape before closing the menu. If the user then CANCELs the *Paste*, control will not return to the toolkit, causing it to leave the *Edit* menu open. Not only is this behavior confusing to the user, it also creates a risk that a buffered button click will start a second *Paste* request if the current request is canceled. The only way to avoid this problem is to return control to the toolkit at N5 before the *Paste* request completes. However, in the design described above the trigger will misinterpret this return, assuming that the request has completed. It appears that changes to the application are needed to avoid this problem.

We learned the following lessons from this work:

- Runtime support for attentiveness requires knowledge of requests that cannot be easily obtained without help from the system's developers
- There are substantial obstacles to implementing replay apart from knowledge of the application
- Mediators must have knowledge of the request that initiated communication with collaborating systems

4.3 General design pattern

Given the obstacles that we encountered in our prototype system, we decided to create a design that would allow us to capture and exploit application-level knowledge of the requests in a system. This design can be used to structure new systems, but can also be retrofitted into existing systems to improve their attentiveness. To avoid spreading responsibility for attentiveness throughout the system, we confined the design to a small number of new classes, shown in Figure 4.8. In this figure the *controller* represents an existing component of the system that is responsible for detecting new requests. In toolkit-based applications the

[6] The poll() and select() system calls provide similar functionality, but were invented by different variants of UNIX. Most UNIX implementations provide both calls to maximize compatibility.

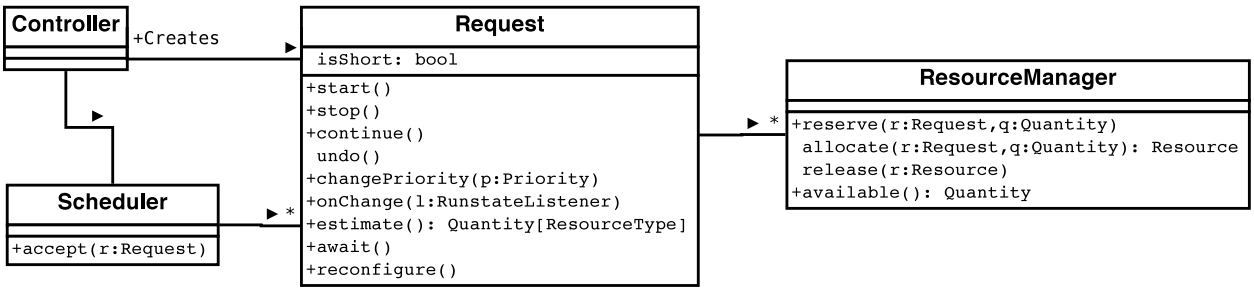


Figure 4.8: This class diagram shows the major classes in our design pattern for attentive systems. Methods marked with a '+' are short.

controller represents the combination of the toolkit and the callbacks registered by the application. These callbacks create new *request* objects and submit them to the *scheduler*. We do not discuss the controller in more detail. In the sections that follow we discuss the three other classes on the diagram–the Scheduler, Requests, and ResourceManagers–explaining how the classes address responsibilities implied by the definition of attentiveness. In the discussion that follows, we prefix short calls with "+." Potentially long calls have no prefix.

4.3.1 The scheduler

The **Scheduler** provides a central point of control for requests in the system. It accepts requests submitted by the controller via the +accept() method. The scheduler decides when to start the request by considering the request's priority, the priorities of other requests in the system, and the availability of resources. After a request has been started, the scheduler continues to track its priority, which may change as the request executes. Changes in request priorities and the arrival of new requests may cause the scheduler to redirect one or more active requests. It does this by invoking methods on individual request objects.

In systems that are not thread-safe, such as Inkscape, the scheduler ensures that at most one request is active in the system at any given time. When this design is applied to these systems there are actually two threads. The scheduler and the toolkit share a thread. The scheduler gains access to this thread when the callbacks registered with the toolkit call the scheduler's +accept() method. The scheduler returns control to the toolkit in a short period of time. A second thread is encapsulated in a request object and executes the request. This thread has access to the application's state, which may not be thread-safe. The classes in this design ensure that the toolkit thread never directly accesses the application's state, and that the application does not access the toolkit's state. The runtime support described in Chapter 5 can enforce this separation of state with minimal impact on the application's performance.

4.3.2 Requests

Request objects serve several purposes in our design:

- Like LISP futures [51], they protect the scheduler from potentially long operations that occur while processing requests. As a result, the amount of analysis needed to ensure that the scheduler will not be blocked is greatly reduced.
- They allow the scheduler to control requests by calling short methods that are part of the request interface.
- Requests provide a way to track and record the system's activities and resource allocations. As a result, requests provide information that can be used to inform the client of the system's current state, inform

the scheduler's decisions to start and stop requests, and aid developers in debugging attentiveness failures

Requests provides up to four short methods that provide low-level mechanisms that the scheduler can use to redirect the request: +start(), +stop(), +continue(), and +undo(). The +start() method begins processing of a request, generally by starting a new thread that is encapsulated in the request. The +undo() method stops the processing of the request and reverses the request's changes, and will typically be implemented with the checkpointing scheme described in Section 5.1. The +stop() method pauses a request, temporarily releasing its resources. However, the request preserves as much of its work as possible, rolling back only the work that could cause other requests to fail if they are started. To ensure that +stop() is short, some systems may make it asynchronous, notifying the listeners registered through the +onChange() method when the request has stopped. It is also possible for systems to implement +stop() by using checkpoints. Finally, the +continue() method resumes execution of a stopped request.

The +changePriority() changes the request's priority and informs the listeners registered with the request (generally the scheduler) of the priority change. This method is short and may be called from any thread. A request's priority may be changed while it is executing. These priority changes generally come from the controller and may cause the scheduler to +stop() or +undo() the request.

Requests provide a short +estimate() method that estimates the request's future resource consumption. In systems with a large number of requests and a small number of resource types the +estimate() method's implementation may be delegated to the resource managers. The scheduler uses the information provided by +estimate() to avoid over-allocation of the system's resources.

Finally, a request can block, waiting for another request to complete by calling the await() method. The +await() method implements priority inheritance [90], raising the target's priority when the caller's priority is higher. This is useful in cases where a high-priority task becomes dependent on a request that was initiated by a lower priority task For example, in Thunderbird a user may attempt to view a message that was originally being downloaded by the junk mail scanner. Since the junk mail scanner is normally a low priority task, the user's attempt to view the message is very likely to raise the priority of the message download. The problem of determining that the junk mail scanner and user are attempting to download the same message is left to the application. In this case the application could provide a hash table to map from message identifiers to request objects.

Figure 4.9 shows the sequence of messages among these components in a typical system. The scheduler gives incoming requests the opportunity to make calls to the toolkit by calling +reconfigure() before returning control to the toolkit. Requests can use this method to make reentrant calls to the toolkit that may affect the interpretation of future events. However, requests must be careful to reverse these calls if the request is canceled in the future by a call to the +undo() method. In addition, the +reconfigure() method must be short to avoid blocking both the toolkit and the scheduler.

Some requests must make toolkit calls as the request is processing. For example, a long-running request may need to make toolkit calls to provide progress feedback to the user. Requests accomplish this by creating new request objects (called subrequests below) and submitting them to the scheduler. By default these new request objects have a dependency on the request that created them. This approach is similar to Swing's [34] invokeLater() call and also Sagas [38], a technique that is used in database systems to avoid conflicts while processing long transactions. The scheduler tracks the dependency between requests and their subrequests, and calls +undo() on the subrequests before +undo() on the request that created them.

When requests use checkpoints to implement the +stop() and +undo() methods the scheduler is responsible for ensuring that a checkpoint is in place before it calls a request's +start() method. The scheduler may use the knowledge provided by the request's isShort flag to improve efficiency, by not establishing a checkpoint before calling these requests. The scheduler may also reuse checkpoints when requests complete quickly. If the scheduler decides to restore the checkpoint, it is responsible for replaying requests completed after the checkpoint was taken. This approach is similar to the way that checkpoints are used to debug long-running

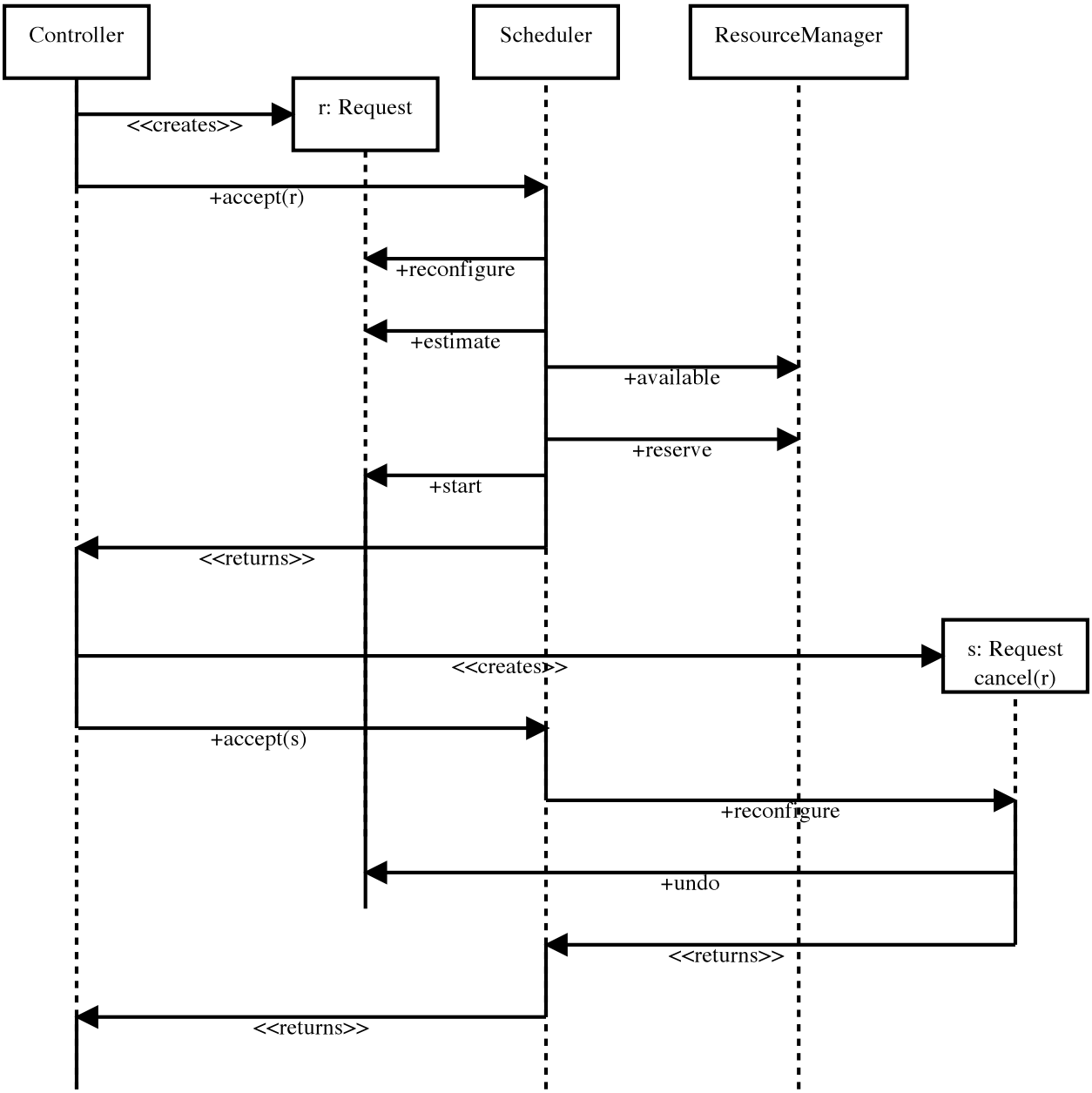


Figure 4.9: This sequence diagram shows the interactions among components when the design is applied in a single-threaded system. Two requests are shown, the seconds request cancels the first request. The first request continues to execute for a brief period of time after the cancel. During this time the scheduler must not submit an additional request.

programs [101]. This design allows the scheduler to limit the overhead of checkpointing at the expense of longer redirection times.

4.3.3 Resource managers

The promptness of systems often depends on the careful management of resources. Resource constrains occur in two of our example systems. In our client-server system, the server must be careful to avoid over-subscription of processors and memory as requests arrive. Thunderbird, described in Section 4.1.3, must carefully manage network resources when communicating with IMAP servers. The diverse nature of these resource constraints argues that we should provide a general framework for managing resources, allowing developers to choose the resources to be managed and the management strategy based on the constraints encountered by their systems. The framework should allow the incremental addition of new resource types, since developers may discover new constraints as they implement and test the system.

We accomplish this by allowing developers to create multiple resource managers, each of which manages a single resource type. Resource managers do not exert direct control over the system. Instead, they track the current level of resource consumption on a per-request basis. The scheduler retrieves this information by calling the short +available() method to determine the availability of the resource type provided by a resource manager. When an exact estimate is not possible in a bounded time, the resource manager may return an approximation to keep the implementation of +available() short.

Resource managers also contain methods that requests use to obtain resources: the allocate() and release() methods. These methods are used only by the request's implementation, and therefore may be long without compromising the responsiveness of the system.

4.4 Applying the pattern to the RMI client and server

We have applied the pattern described above to the RMI-based client-server system. Both the client and the server have request, scheduler, and resource manager classes. Request objects on the client and server have similar responsibilities.

The schedulers on the client and the server have different roles. The scheduler on the client uses the +start(), +stop(), and +undo() methods to manage the client's resources, including server connections and network bandwidth. In addition, it publishes an interface that can be polled by the scheduler on the server to determine the state of individual requests. The server running on the client notifies the server of this interface by placing a backpointer into each request before forwarding it to the server's scheduler.

The scheduler running on the server selects the requests to start to achieve a fair assignment of resources to competing clients. In addition, the scheduler uses the back-pointer in requests to periodically poll the client that submitted the request to confirm that the request has not been abandoned. Polling must be isolated from the scheduler's other activities to ensure that the scheduler is not blocked. It may appear to be simpler to rely on the clients to send redirection notifications to the server. However, this would make the server vulnerable to crashed clients, network partitions, and denial of service attacks from malicious clients.

We applied this modified pattern to our simple RMI client-server system. We did not use checkpointing to implement the request's +stop() and +undo() methods. Instead, we implemented +undo() by invoking the deprecated Thread.stop() method. We attempted to protect RMI from the effects of Thread.stop() by moving the request's processing into a separate thread. This thread returns a single value to the RMI thread via a volatile field in the Request object. Since stores to volatile fields establish a happens-before relationship in Java, this approach greatly reduces the chances that an RMI thread could encounter a corrupted object. The RMI thread will either see a null pointer, indicating that the request was canceled before producing a result, or a non-null pointer to an object containing a complete result.

We configured the scheduler on the server to poll for disconnected clients every 100ms. The resulting server was capable of canceling abandoned requests promptly. The time to compute the first 15,000 digits of π increased from 10.53s to 11.03s. The added overhead was largely due to the polling. Decreasing the polling rate to once every 1000ms yielded a computation time of 10.66s. We did not observe failures due to state corruption due to the use of Thread.stop() during our testing, but would not recommend this approach for production systems.

4.5 Conclusion

In this chapter we have considered several designs that pose a risk to attentiveness:

- In Inkscape restrictions on the use of threads due to non-thread-safe code increase the risk of blocking, since designers are not free to use threads to isolate the effects of long operations
- Thunderbird and RoundCube abandon requests pose a risk to attentiveness in servers when their requests exhaust constrained resources. Protocols like IMAP, and to some extent RMI, may contribute to this problem by forcing clients to abandon requests in order to remain attentive.
- Thunderbird executes background tasks in parallel with requests from the user and fails to prioritize the assignment of assign constrained resources to user requests.

We have also identified strategies that support attentiveness:

- Reservation of resources for both the user interface and also the user's requests.
- Limiting the scope and number of background tasks.
- Tracking all of the tasks that are active in the system, including background tasks.

We incorporated these strategies into a design of that supports attentiveness and applied the design to a simple client-server system. The use of this design overcomes the limitations of the RMI protocol, resulting in measurable improvements in attentiveness.

However, runtime support for redirection is needed to apply this design safely to more complex systems. We describe this runtime support in the next chapter.

Chapter 5

Runtime support for attentive systems

Attentive systems must be able to redirect requests in progress. The system must preserve relationships among multiple parts of the system's state while redirecting requests, including:

- The threads in the system
- Persistent state such as files
- The system's memory
- Requests that have been submitted to collaborating systems

The relationships among different parts of the system's state are often not apparent when examining the system's implementation. In addition, relationships often occur among modules maintained by different developers, making it difficult for a single developer to reason about redirection. This chapter describes the design and partial implementation of a runtime to manage these dependences. The runtime uses the directives described in Chapter 3 to capture intent that cannot be inferred from the system's implementation, and uses this intent to implement the per-request start(), stop(), continue(), and undo() operations described in Chapter 4.

The soundness of redirection depends on the accuracy of directives. If the directives do not describe the system's actual behavior, the runtime may break some of the relationships among the parts of the system's state, potentially leading to data corruption and/or the eventual failure of the system. To address this problem the runtime is able to verify that the information provided by some directives matches the system's behavior as it executes. This validates the directives *for the current execution* of the system. Dynamic checking can be very expensive. For example, Helgrind [89], a dynamic checker that detects data races, increases the running time of systems by a factor of 20-1300. However, our runtime is able to check directives that rule out data races while increasing the running time by only a factor of 3. When the runtime encounters an inaccurate directive it halts the system and generates a report. The report contains both a reference to the inaccurate directive and also a description of the behavior that was not accurately described by the directive. Developers can then resolve the inaccuracy either by modifying the directive or changing the implementation of the system to preclude the event.

Two runtimes are described in two sections. Section 5.1 describes the design of a runtime that is able to redirect requests. The runtime described in the first section has not been implemented. Section 5.2 describes the design and implementation of a runtime that is able to identify inaccurate directives. It quantifies the performance impact of checking directives by adding directives to benchmarks taken from the PARSEC 2.0 suite [13]. These directives are sufficient to prove that the benchmarks do not encounter data races during execution. The work described in this section was done with help from Andrew Wesie. Section 5.3 concludes by proposing additional work that would improve the design and implementation of the runtimes.

5.1 Runtime support for redirection

The discussion in this section uses a simplified IMAP server as a running example. IMAP servers manage multiple folders containing email messages. Multiple clients connect to the server and can access the same folder simultaneously. As a result IMAP servers must preserve three forms of consistency when operations are redirected:

- Consistency within the IMAP server
- Consistency between the IMAP server and its clients
- Consistency among the clients

Systems must often employ detailed knowledge of communication protocols to preserve the second and third forms of consistency. The runtime does not directly incorporate this knowledge. Instead, the runtime uses mediators, special components registered by the developers of the system, to handle redirection. Mediators are described in Section 5.1.4.

IMAP servers often use multiple threads to support concurrent processing of requests from different clients. The discussion below assumes that the server has four threads, providing examples of the risks to consistency that occur in threaded systems. The example assumes that the IMAP server follows the active object pattern, where each thread has exclusive access to a defined part of the system's state called a region and communicates with other threads using asynchronous messages. The four threads have different responsibilities:

T_C handles communication with the client that initiated the *Copy*.

T_O handles communication with the second client, which observes the destination folder as the copy progresses.

T_S manages the folder that is the source of the messages.

T_D manages the folder that receives the new messages.

The server maintains persistent state in four files:

F_S holds the messages in the source folder.

F_D holds the messages for the destination folder.

F_L holds a log of requests from the user.

F_I holds a table with one record per folder giving the next valid UID, the unique identifier assigned to messages when they are placed in a folder.

The system's memory consists of three data structures:

M_S maps each message id in the source folder to an offset and length in F_S. When the server restarts this map is reconstructed by scanning the file. The system must ensure that M_S and F_S remain consistent during redirection.

M_D maps each message id in the destination folder to an offset and length in F_D. When the server restarts this map is reconstructed by scanning the file. The system must ensure that M_D and F_D remain consistent during redirection.

M_I caches the information in F_I, allowing the server to quickly assign messages identifiers to new messages in folders. M_I is loaded from F_I when the server starts, and F_I is kept in sync with the view in M_I.

M_L represents memory that is controlled by the runtime libraries on the system. The developers of the system do not control these libraries and are unaware of their implementation.

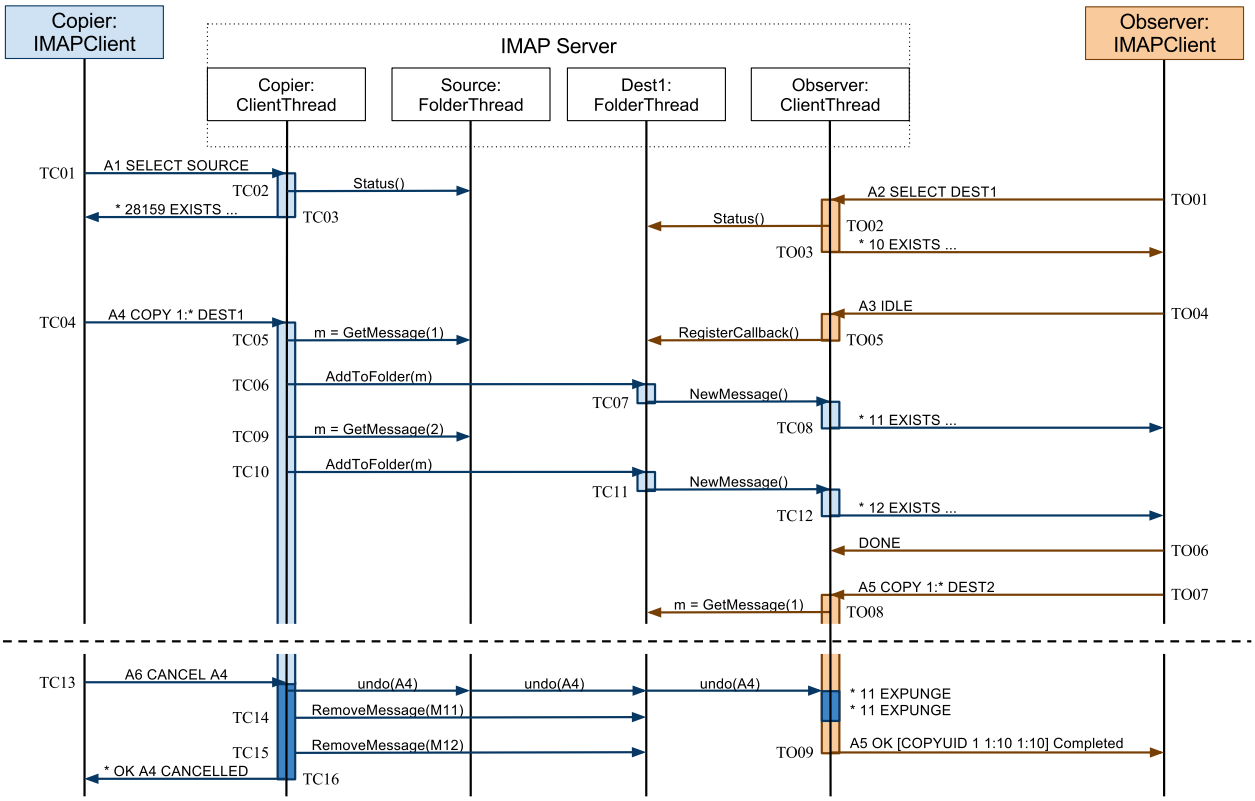


Figure 5.1: Canceling requests in a server can involve negotiation among its clients. When the copy request marked with an A4 is canceled, the copy request marked with A5 must be modified to exclude messages 11 and 12.

In an actual system there would be additional data structures, including memory that is dedicated to each of the threads. For simplicity the following discussion does not address these data structures. The consistency of these data structures can be maintained using the techniques described below.

The IMAP server allows clients to observe the progress of long running commands submitted by other clients. Figure 5.1 shows the interactions between the IMAP server and two clients. The first client initiates a copy command to copy messages in a source folder to a second folder called the destination folder (A4 in the figure). The second client observes the destination folder, and initiates a second copy (marked with A5) while the first client's copy is in progress. A5 references two messages (11 and 12) that were placed in the destination folder by A4. As a result, A5 is also affected when the copier cancels A4. The runtime preserves the internal consistency of the server and also notifies clients of the changes. For example, its reply to A5 indicates that only 10 messages were copied and that messages 11 and 12 have been removed from the folder.[1] Many IMAP servers, including Dovecot [27] and Cyrus [24], implement full isolation of requests as they execute. Isolation simplifies the design of the server, but introduces other problems. For example, isolation makes it impossible to provide progress feedback for long running commands.

The discussion follows the issues that arise when A4 is canceled. Section 5.1.1 discusses the features of the runtime that allow developers to identify new requests. Section 5.1.6 describes the features of the runtime that allow the first client's copy request to be stopped without compromising the system's consistency. To do this, the runtime must undo the request's changes to M_D and M_L while preserving the changes is M_I. The

[1] The diagram shows that message 11 was removed twice. This follows the conventions of the IMAP protocol, where message 12 is renumbered to 11 when message 11 is removed.

runtime support for managing this process is discussed in Section 5.1.2. In addition, the runtime must undo the changes that the copy request made to F_D, removing all of the messages that were added to the file by the copy. Support for making these changes, while retaining the changes in F_L, is discussed in Section 5.1.4. The runtime must also restart the second copy operation, since it observed some of the changes made by the first copy. To do this, it tracks dependencies among requests, as described in Section 5.1.3. Section 5.1.7 describes an approach to supporting black-box composition with modules by using tollgates. Finally, Section 5.1.8 describes solutions to thread life-cycle issues problems that can occur while rolling back requests.

5.1.1 Request tracking

The runtime is not able to identify new requests without help from directives. Requests can arrive through many communication mechanisms, including network sockets, pipes, files, and signals. Systems may use the same mechanisms for background processing that is not directly tied to requests. In addition, in toolkit applications the identification of new requests occurs after the communication in a series of callbacks, as discussed in Chapter 4.

Therefore, the runtime relies on the directives shown in Listing 5.1 to identify requests. When a thread invokes the create_request() directive, the runtime creates a new request and returns a request_t object that can be used to reference the request in future directives.

The runtime assumes that the processing for a request happens on one or more threads. The relationship between threads and requests is dynamic. An incomplete request may be associated with 0, 1, or many threads at once. Threads are always associated with a single request. The request that is associated with a thread may change during the thread's lifetime. New requests are not associated with threads. New threads are initially associated a special request called the nullRequest. The nullRequest cannot be redirected. The nullRequest allows developers to identify activities in the system that are not associated with a user-generated request, such as the processing that occurs in a GUI toolkit as it identifies a new request.

Developers indicate that a thread is about to do work for a request by invoking the associate_request(request) directive. Developers can retrieve the request currently bound to a thread by invoking the current_request() directive. Developers may need to retrieve the current request to allow additional threads to bind to the request. For example, in Figure 5.1 ClientThreads would pass the current request when sending messages, such as AddToFolder, to FolderThreads.

Developers indicate that the request currently bound to a thread has completed by calling complete_request(). By calling this directive the thread is asserting that there are no other threads associated with the request and that no thread will call associate_request(current_request()) in the future. Code that violates either of these conditions uses the directives inconsistently, and will cause the runtime to report an error.

5.1.2 Tracking changes to memory

Requests modify memory as they execute. For example, the COPY request at A4 in Figure 5.1 modifies M_D when it adds messages to the Dest1 folder, adding entries for the messages that point to their location in F_D. In addition, it modifies M_I as it assigns identifiers to the messages that it adds. Finally, the libraries that are used to access and update the files may make changes to M_L. The runtime tracks these changes, associating each with the request that caused it. It also ensures that the changes can be reversed if the undo() operator is invoked on the request. To do this, the runtime splits the address space of the system into a series of regions. Threads, executing on behalf of requests, invoke directives to gain read and/or read-write permissions to regions before accessing them, as described in Chapter 3.

To avoid unacceptable overhead the runtime does not examine individual memory accesses. Instead, it relies on knowledge of the access policies that protect regions. These policies are provided by the directives, and are discussed in detail in Section 3.3. For example, multiple FolderThreads may attempt to access M_I.

```
typedef enum {
   ready, running, stopping, stopped, completed, finalized
} state_t;

typedef struct {
   map<region_t *, generation_t> readSet;
   map<region_t *, writeEvent_t> writeSet;
   set<block_t *> deferredFrees;
   state_t state;
   bool isShort;
   int threadsBound;
} request_t;

set<request_t *> allRequests;

__thread request_t *currentRequest;
__thread request_t *nullRequest;

request_t *create_request(bool isShort) {
   request_t *rval = new request_t(isShort);
   rval.state = stopped;
   rval.threadsBound = 0;
   allRequests.add(rval);

   return rval;
}

void associate_request(request_t *request) {
    exit_as_writer(currentRequest, threadRegion);
    currentRequest.threadsBound--;
    if (request.state != stopping) {
       request.threadsBound++;
       enter_as_writer(request, threadRegion);
       currentRequest = request;
       request.state = running;
    } else {
       thread_stop();
    }
}

void complete_request(request_t *nextReq)
{
   request_t *comp = currentRequest;
   associate_request(nextReq);
   if (comp.threadsBound) {
      report_error("Attempting to complete with multiple bound threads");
      return;
   }
   if (comp.state == stopping)
       completed_rather_than_stopped();
   comp.state = completed;
}
```

Listing 5.1: The runtime uses this data model to track requests. The __thread keyword is a storage modifier, supported by GCC, that indicates that the variable should be placed in thread-local storage. The map and set types are place-holders for abstract data types that provide a subset of the functionality of the corresponding C++ templates. The operations given above are short and should be considered to be atomic.

To avoid data races the implementation of M_I may use locking to ensure that at most one thread accesses the region at a time. When a thread obtains the lock the runtime will assume that the thread is about to write to the region. The runtime will ensure that the thread's writes can be reversed if its request is redirected. When the thread releases the lock the runtime will assume that the thread will no longer access the region. Unlike transactional memory systems, the runtime must still be able to roll back the thread's changes. Changes are committed only when the request is finalized, as described in Section 5.1.5.

In other cases a single thread may have exclusive access to a region. For example, M_D is accessed only by T_D, the FolderThread that controls the Dest1 folder. In this case the runtime assumes that any request that is bound to T_D writes to the region.

The discussion below covers three aspects of the support for tracking changes to memory: identifying regions, making copies of regions before they are modified by requests, and managing the address space of the system to ensure that the content regions can be restored during the undo() operation. Tracking changes to regions relies on the functionality for tracking dependencies described in Section 5.1.3 for part of its functionality.

5.1.2.1 Defining regions

Attentive systems may incorporate code that is not aware of regions. In addition, standard memory allocation routines have no concept of regions. Therefore, the runtime constructs regions indirectly, as the system executes. Directives embedded within the system specify which region should hold new blocks that are allocated by a thread. The region that will hold future allocations can be changed by invoking the bind() directive. The implementation of bind() saves the region in thread-local storage, making it available when the memory allocation routines are invoked.

The runtime intercepts low-level calls that allocate memory, including malloc(), calloc(), realloc(), mmap(), and mmap64(). It intercepts most calls by using the LD_PRELOAD environment variable to reroute calls to these routines to versions provided by the runtime. However, libc, the library that provides system call support and memory management for most systems, occasionally calls memory allocation routines directly, bypassing the intercepts. Therefore, the runtime also intercepts calls by placing a jump instruction at the beginning of some of these routines.

The runtime also updates data structures that allow addresses to be translated back to the region that contains them. These data structures support the region_of(block_t *) directive, which retrieves the region containing a block of memory.

5.1.2.2 Preserving the contents of regions

The runtime system makes a copy of a region the first time that a request gains access to the region. When the undo() operation is invoked the runtime restores the content of the region from this copy. The runtime supports two different copying approaches: an eager copy and a lazy copy. The eager copy is relatively simple, but can be slow for large, sparsely updated regions. In addition, it consumes the address space of the system. In the worst case each non-finalized request could hold a copy of every region in the system.

The eager copying scheme follows the following steps:

1. The runtime checks for another request that is already writing to the region. If another request is writing to the same region it will already have a copy of the region. The runtime avoids creating a redundant copy by creating a two-way dependency between the requests. As a result, issuing an undo() operation with either request will roll back both requests.
2. If no other request is writing to the region, the runtime examines the region, calculating the number of blocks and the total size of the blocks.

3. It allocates a block of memory that is large enough to hold the address, size, and current contents of each block within the region.
4. Finally, it copies the blocks in the region, noting their size and original address.

The primary overhead in the eager scheme is due to the copies of the region's blocks. On an Intel Xeon E5405 CPU running at 2.0GHz, copying 660MB of memory takes approximately 0.1s. Therefore, the process of creating a checkpoint on these systems is potentially *long*, using the terminology outlined in Chapter 2. When systems use the design pattern outlined in Section 4.3 the overhead for the copying will not affect the acknowledgment time of the system.

The lazy copying scheme achieves better performance for large, sparsely written regions by postponing the copy operation. It does this by establishing a copy-on-write mapping to the region. The mapping removes the system's write access to the region. As a result, any attempt to write the region will create a fault. The kernel intercepts the fault, copies the page being written, which is generally 4k of memory, grants the process read-write access to the page. Finally, it allows the write to proceed.

Lazy copying is advantageous for large regions that are updated sparsely. The mappings can be established quickly, as shown in Figure 5.2. However, if every page in the region is written the added faults will increase the copying time for the region by about 40% when compared to eager copying, as shown in Figure 5.3. In addition, lazy copying can preserve at most one copy of a region at any given time. Therefore, the runtime must introduce additional dependencies among requests to ensure that undo() operations will not compromise the consistency of the system.

The runtime can implement lazy copying by using standard system calls. The implementation relies heavily on memory mapped files. It creates a memory mapped file in the tmpfs filesystem to hold the last finalized version of the system's memory. When a thread obtains write permission to a region, the runtime executes the following operations:

1. It checks if another unfinalized request is already modifying the region. If so, it creates a bidirectional dependency between the requests and stops. The dependency will ensure that the region is restored from the clean copy if either request rolls back.
2. It calls mmap() with the MAP_PRIVATE flag for each of the pages in the region. Once this call has completed, writes to the pages will be reflecting in the process's memory space but will not affect the pages for the region in the file in tmpfs.
3. During an undo() operation the runtime issues a second mmap() for each of the pages in the region. The mmap() call will discard any modified pages, restoring the region from the clean copy in the tmpfs filesystem.
4. If all of the requests that have modified the region have finalized, the modified pages are now the clean version of the region. The runtime commits these changes by using the pwrite() system call to write the pages to the file in tmpfs. Ideally the kernel would optimize this write, skipping any pages that have not been modified. Once these calls are complete, the modified pages can be discarded by issuing a second mmap() call.

To use lazy copying the runtime must modify the way that memory is allocated so that a page belongs to at most one region. This change can be added to the routines that maintain the mapping between regions and addresses.

The application of copy-on-write to checkpointing problems is not novel. Copy-on-write has been used in the past to support I/O prefetching [35] and to capture snapshots of a system's state before committing them to disk [41]. However, this technique is well-suited to attentive systems, since it is lightweight relative to the systems' promptness requirements and can be employed in systems without engaging in extensive analysis of every module used to construct the system.

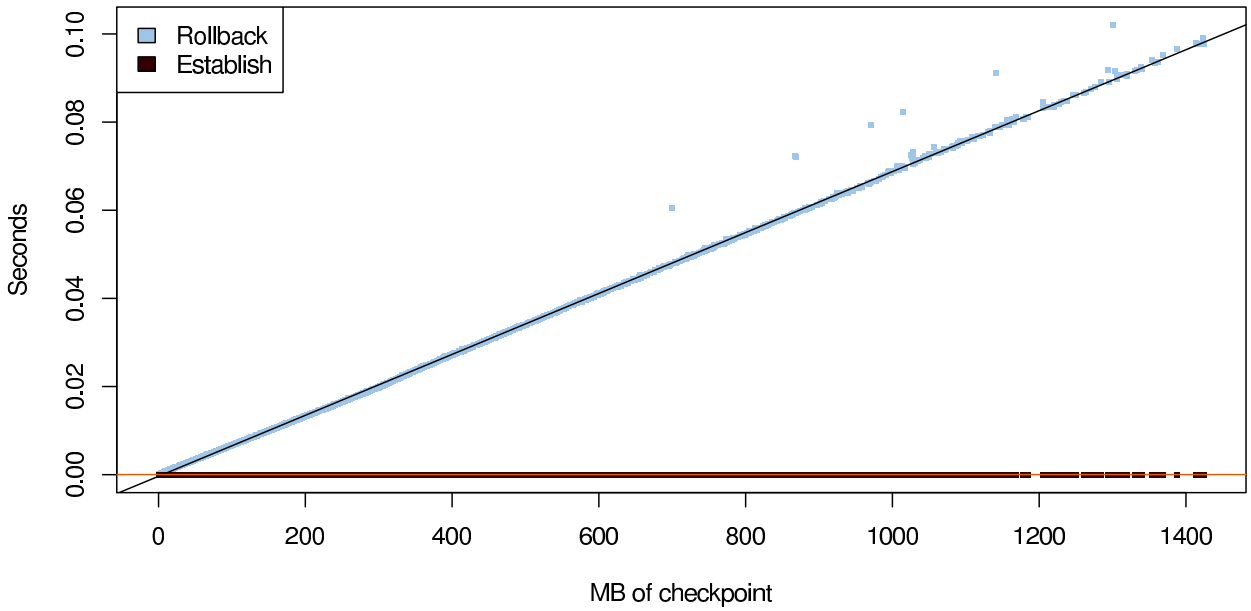


Figure 5.2: The cost of establishing a lazy checkpoint on an Intel Xeon E5405 CPU running at 2.0GHz with 4GB of memory.

5.1.2.3 Postponing free()

The implementations of the undo() operation for both eager and lazy copying assume that regions can be restored at their original addresses. Therefore, the runtime must ensure that a region's addresses are not reassigned to a second region. If addresses were reassigned, the undo() operation would not be able to restore the first region modifying the second region.

To avoid this problem, the runtime intercepts calls to free(), realloc() and munmap(). When there is a copy for the region, frees of blocks in the region are placed in the deferredFrees field of the active request. The request will complete these frees when it is finalized. It may be possible to improve on this design by applying one of the ideas discussed in Section 5.3.3

5.1.3 Dependency tracking and undo()

Dependencies arise in attentive systems when requests observe the partially completed work of other requests during their execution. For example, there are both external and internal dependencies between the COPY request marked with A5 in Figure 5.1 and the COPY request marked with A4:

- The internal dependency is created when A5 accesses regions that have been modified by A4, including M_D, M_I, and M_L. The discussion in this section addresses internal dependencies.
- The external dependency is created when Observer receives the "11 EXISTS" message at TC08 and the "12 EXISTS" messages at TC12. These messages are generated as A4 executes, and are no longer valid after A4 is canceled. Dependency tracking is able to automatically detect this dependency, but is unable to restore the consistency of the system without detailed information about the IMAP protocol. Therefore, the runtime contacts a mediator, which generates the appropriate IMAP notifications to inform the observing IMAP client of the IMAP server's new state.

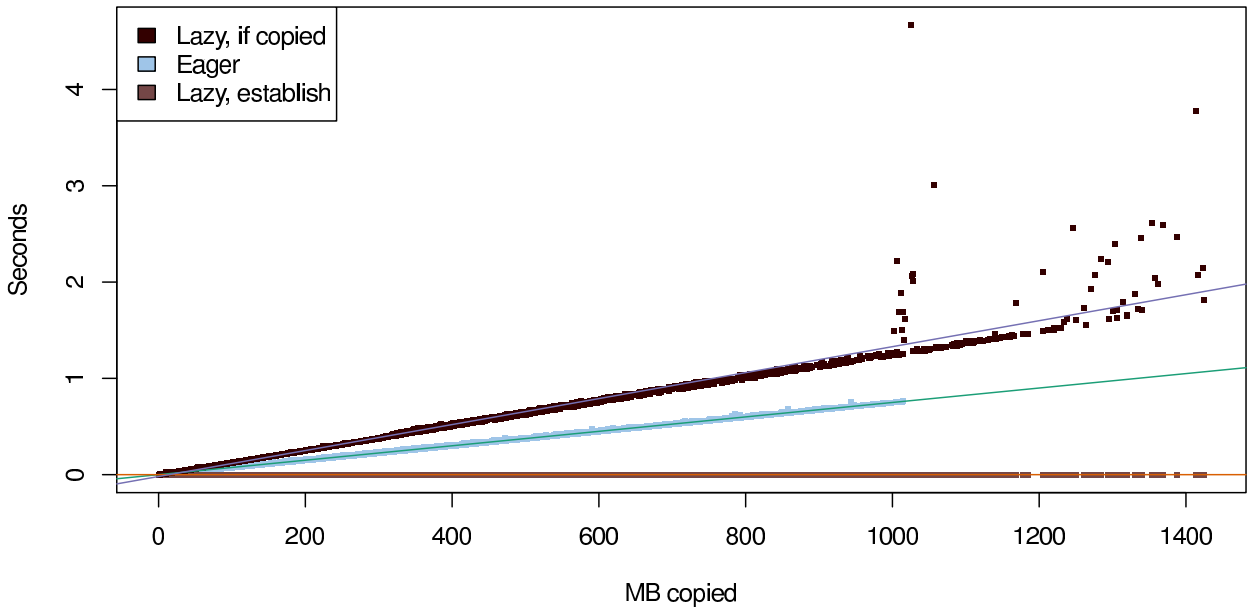


Figure 5.3: Cost of establishing and maintaining checkpoints on an Intel Xeon E5405 CPU running at 2.0GHz with 4GB of memory. The boxes show the results of individual tests. The lines represent a linear fit of the tests. The cost of establishing a lazy checkpoint is low and is represented by the bottom line. The middle line shows the cost of establishing an eager checkpoint, which immediately incurs the cost of copying the data in the checkpoint. The top line shows the cost of preserving a lazy checkpoint when data is modified.

Dependency tracking does not examine individual access to the regions. Instead, it uses knowledge extracted from the directives that threads invoke to gain and release permission to access regions.

The runtime uses information provided by the directives to maintain a generation number for each region. Pseudo-code for the data structures involved is shown in Listing 5.2. The runtime increments the generation number for a region every time that a thread obtains write access to the region. The generation number acts as a version number for the content of the region, with higher numbers corresponding to more recent version.

To illustrate, consider the generation number of M_D, the region controlled by the Dest1 FolderThread in Figure 5.1. Assuming that the initial generation of M_D is 0,[2] the generation of M_D, as viewed by the region and T_D, will become 1 at TC07, 2 at TC11, and will return to 0 as a result of the undo() operation invoked by the CANCEL at TC13.

Requests become aware of a region's current generation number whenever a thread bound to the request invokes a directive to gain or release permission to access the region. The readSet of the request always contains the highest observed generation number. For example, A2 observes that the generation number of M_D is 0 at TO02. A3 observes the same generation number at TO05. A4 detects that the generation number is 1 at TC06 and updates it to 2 at TC10. In addition, at TC06 A4 modifies M_D for the first time. Therefore, it copies the region and saves the copy of the region with a generation number of 0 to its writeSet. This is the copy that is restored during the undo() operation at TC13.

A5 discovered that the generation number of M_D is 2 at TO08, and records this number in its readSet. During the undo() of A4, shown at TC13, the runtime restores of M_D from A4's writeSet, dropping the generation number of M_D to 0. It detects that A5 depends on A4 when it examines A5's readSet and discovers that A5 observed a later version of M_D than is now current. The runtime responds by also rolling

[2] Any initial value will work.

```
typedef struct {
    bool createsDependencies;
    generation_t generation;
    int writers;
    request_t *oldestRequestWithCheckpoint;
} region_t;

typedef struct {
    checkpoint_t *chkpt;
    generation_t time;
} writeEvent_t;

region_t tollgateRegion;

__thread region_t *threadRegion;

void new_thread() {
  threadRegion = new_region(THREAD_LOCAL);
  enter_as_writer(threadRegion, pthread_self());
}

region_t *new_region(policy_t policy)
{
  region_t *rval = new region_t;
  rval.createsDependencies = (policy != THREAD_SAFE);
  rval.generation = 0;
  enter_as_writer(currentRequest, rval);
  return rval;
}
```

Listing 5.2: Request-level dependency tracking

back A5's changes. Allowing A5 to continue without rollback could compromise the consistency of the system, since the contents of M_D changed. The runtime invokes undo() on A5 to undo A5's observation of the later state of M_D.

The runtime shares many features with some implementations of software transactional memory (STM) [26]. Like STM, the runtime uses generation numbers to detect dependencies. However, it differs in the following ways:

- STM libraries differ because they isolate transactions, and therefore must check for conflicts among transactions whenever a transaction commits. The dependency tracking in the runtime is invoked only when the undo() operation is invoked during redirection. The runtime assumes that redirection will be relatively rare. Therefore, it has been designed to reduce the cost of tracking dependencies at the expense of doing more analysis during redirection.
- Requests are not isolated. Therefore, the runtime is unable to apply certain optimizations that are used in STM implementations. For example some STM systems use a global generation counter, incremented on each commit, to eliminate the tracking of read sets for short read-only transactions. The STM system assumes a conflict if this counter updates and restarts the read-only transactions.
- The runtime uses directives to collect dependencies at the level of regions rather than individual memory locations. This greatly reduces the cost of collecting dependencies: there is no need to collect information about each memory access. However, the runtime will detect false dependencies among requests that access different parts of the same region, potentially increasing the expense of redirection. Developers can reduce this problem, at the expense of writing more directives, by using smaller regions. In addition, the runtime can fail to detect dependencies if the directives are inaccurate.

```
void no_region_dependencies(region_t *region) {
  region.createsDependencies = false;
}

void enter_as_reader(request_t *request, region_t *region) {
    if (region.createsDependencies)
      request.readSet[region] = region.generation;
}

void exit_as_reader(request_t *request, region_t *region) {
    if (region.createsDependencies)
      request.readSet[region] = region.generation;
}

void enter_as_writer(request_t *request, region_t *region) {
    if (region.createsDependencies) {
      enter_as_reader(request, region);
      generation_t checkpointGeneration = region.generation;
      region.generation++;
      if (!request.writeSet.has(region))
        request.writeSet[region] = writeEvent_t(checkpointGeneration, makeCheckpoint(region));
      region.writers++;
  }
}

void exit_as_writer(request_t *request, region_t *region) {
  if (region.createsDependencies) {
     request.readSet[region] = region.generation;
     region.writers--;
  }
}

checkpoint_t *makeCheckpoint(region_t *region) {
  if (lazy_copy(region) &&
       region->oldestRequestWithCheckpoint &&
       region->oldestRequestWithCheckpoint->state != finalized) {
     // The region holding the checkpoint now depends on current_request()
     region->oldestRequestWithCheckpoint->readSet[region] = region->generation;
     return NULL;
  }
  // Checkpointing code goes here
}
```

Listing 5.3: Operations that are invoked during permissions changes

The subsections that follow discuss the detailed design of the runtime. Listing 5.3 describes operations that modify the request and region data structures when threads obtain access to regions. Section 5.1.3.2 discusses the detailed design of the undo() operation.

5.1.3.1 Operations for permissions changes

The detailed design of the operations that track dependencies is given in Listing 5.3. Developers can specify that regions do not propagate dependencies by using the no_region_dependencies(region_t *) directive. For example, developers may do this for the default thread-local regions associated with worker threads in the map phase of a MapReduce [25] system. By invoking this directive the developers are asserting that the workers do not retain state after they process each element in the input set. This assumption is reasonable for worker threads involved in the Map phase, but may not apply to workers doing the Reduce. Developers must be very cautious when invoking this directive: the runtime has no way of detecting inconsistent use of the directive.

The enter_as_reader(request, region) operation uses generation to detect dependencies among requests. It does this by saving the current value of the region's generation field to the request's readSet. It does not check for an existing entry in readSet before saving the value. Any existing entry would either refer to an equal value of generation, making the update a noop, or would refer to a lower value of generation. By updating generation to the latest value, enter_as_reader() preserves all of the request's previous dependencies while potentially adding new ones.

The enter_as_writer() operation assumes that every writer of a region is also a reader of the same region. Therefore, enter_as_writer(request, region) starts by calling enter_as_reader(request, region). Next, it checks the request's writeSet for a reference to the region. Unlike the readSet, existing entries in the write set must be preserved, since they contain a checkpoint created before the request made changes to the region. Replacing this checkpoint with a later checkpoint would make it impossible to undo the request's changes to the region. If the region is not in the writeSet, enter_as_writer() creates a checkpoint for the region, saving the checkpoint and the current value of generation in a new writeEvent_t structure in the writeSet. Finally, enter_as_writer() increments the region's generation field, indicating that the request may modify the region.

Since threads automatically obtain read-write access to regions when they create them, the new_region(policy) directive calls enter_as_writer(currentRequest, newRegion) after it creates a new region. It also initializes the generation field to 0 and provides the appropriate default for the createsDependencies field.

When a new thread is created the runtime creates a symbolic thread-local region to represent the thread's local state, including its stack and registers and any thread-local regions later created by developers. By default this local region will propagate dependencies among requests associated with the thread, as shown in the associate_request() directive in Listing 5.1. Developers can override this behavior by calling no_region_dependencies(threadRegion).

The exit_as_reader() and exit_as_writer() operations update the request's readSet when it releases access to a region. These updates address dependencies that occur when threads write regions while other threads have access to the region. They ensure that the other threads create a dependency on the concurrent writers. During undo() the runtime may need to take additional steps to detect these dependencies.

The partial design of makeCheckpoint() shows the support needed to propagate the dependencies for lazy copies. The approach proposed for lazy copies in Section 5.1.2.2 can support at most one copy per region. The earliest copy for a region must be preserved to ensure that all of the active requests in a system can be rolled back. Therefore, the code in makeCheckpoint() first checks for a request that has a copy of the region. If such a request exists, the code makes the request holding the lazy copy dependent on current_request(). The dependency ensures that the request holding the copy will be rolled back when current_request() is rolled back. Once the dependency has been established there is no need to create a checkpoint. Therefore makeCheckpoint() returns immediately after establishing the dependency.

5.1.3.2 Implementing the undo() operation

A design for undo() is shown in Listing 5.4. The undo() operation is implemented in two phases. The first phase identifies the set of regions and requests that must be rolled back to preserve the system's consistency. In the worse case, this algorithm will execute $O(n^2)$ operations, where n is the number of non-final requests in the system. However, this case will be produced only if each non-final request depends on exactly one other non-final request and the algorithm processes the requests in the worst possible order.

The algorithm begins by restoring the checkpoints of the request being redirected without accounting for dependencies, rolling back the generation values of the affected regions. Next, the algorithm examines the set of non-final requests, comparing the values of generation saved in their readSets to the values in the regions. If the readSet indicates that the request depended on a later version of the region, the non-final request must also be rolled back. The algorithm proceeds by calling simple_undo() on the request and

```
void simple_undo(map<region_t*, checkpoint_t *> *deferredRestores, request_t *req) {
    foreach ((region_t *region, writeEvent_t ev) in req.writeSet) {
        if (region.generation > ev.time) {
          region.generation = ev.time;
          (*deferredRestores)[region] = ev.chk;
      }
    }
    req.readSet.clear();
    req.writeSet.clear();
    req.deferredFrees.clear();
}

void undo(request_t *req) {
    stop_the_world();

    map<region_t*, checkpoint_t *> deferredRestores;
    simple_undo(&deferredRestores, req);
    req.state = stopped;

    queue<request_t *> toCheck;
    queue<request_t *> checked;
    foreach (request_t *request in allRequests) {
        if (request.state != finalized) {
            toCheck.add(request);
        }
    }
    while (request = toCheck.removeFront()) {
        bool addToChecked = true;
        foreach ((region_t *region, generation_t t) in request.readSet) {
            if (region.generation < t) {
                simple_undo(&deferredRestores, request);
                request.state = ready;
                toCheck.add(checked);
                checked.clear();
                addToChecked = false;
                break;
            }
        }
        if (addToChecked) {
          checked.add(request);
        }
    }
    for ((region *region, checkpoint_t *c) in deferredRestores) {
        restoreCheckpoint(c, region);
    }

    start_the_world();
}
```

Listing 5.4: Pseudo-code for request undo

restarts by appending the queue of checked requests to its queue of requests to be checked. If the request being examined appears to be consistent, the algorithm adds it to the queue of checked requests. By placing it in this queue the algorithm ensures that the request will be rechecked if another request rolls back before the algorithm terminates.

During the second phase the runtime restores the content of the regions from the copies identified in the first phase. Eager checkpoints can be restored by using memcpy() to copy the memory. Lazy checkpoints are restored by calling mmap() on the pages in the region with the MAP_SHARED flag. As a result of this call, the modified versions of the pages in the region are abandoned in favor of the original versions. Undo of lazy checkpoints can be highly efficient, as shown in Figure 5.2.

5.1.4 Handling external dependencies

There are two forms of external dependencies in the IMAP server. First, the content of the files affected by A4, including F_D, must be restored when A4 is canceled. Second, the IMAP server must inform the Observer that messages 11 and 12 have been deleted. The Observer became aware of these messages at TC08 and TC12, but the IMAP server no longer knows about the messages after the redirection at TC13.

Unlike internal dependencies, the reconciliation of external dependencies must account for the semantics of operations. For example, in the running example it is sufficient to truncate F_D to its length before A4 began to run. However, if a second copy were modifying F_D, the tail of the file would have to be carefully rewritten to preserve the second copy's messages. F_L, the log of operations on the IMAP server, should not be modified with A4 is redirected, even though A4 modified the file. In addition, the IMAP protocol specifies that the server must not reuse the UIDs assigned to messages 11 and 12 at any point in the future. As a result, the server must be careful to avoid rolling back M_I and F_I.

The strategy for resynchronizing the Observer depends on the semantics of the IMAP protocol and the state of the system. In the running example the IMAP server is able to resynchronize by sending compensations [15], in this case two "*11 EXPUNGE" messages. The mediator is able to generate these messages only because it has detailed knowledge of the previous messages sent to the Observer and knowledge of the protocol.

The runtime allows developers to create components, called mediators, that manage these issues. Mediators can participate in rollback system described above by creating special regions. The rollback system will then use callbacks to invoke the mediator during the operations described in Listing 5.3, and also invoke the mediator to restore the state of the region during undo(). In addition, the mediators are capable of intercepting, and possibly modifying, operations that could initiate communication or modify persistent state on the system. These operations are called subrequests. Mediators may choose to pass through or modify subrequests. The approaches described below are similar to the ones used in transactional memory systems [54, 84].

Some subrequests do not require modification. For example, the "* 11 EXISTS" TC08 can be passed through immediately, since there is always a corresponding "EXPUNGE" that reverses its effect on the Observer.

Other subrequests cause changes that are difficult or impossible to reverse. For example, IMAP clients can initiate a large number of message deletions in a folder with the EXPUNGE command. Normally, an observer in the same folder would see each message as it was deleted. However, the IMAP protocol does not provide a way to reverse these changes if the EXPUNGE is canceled. In this case the mediator may choose to isolate the EXPUNGE command, informing the Observer only after all of the messages have been deleted.

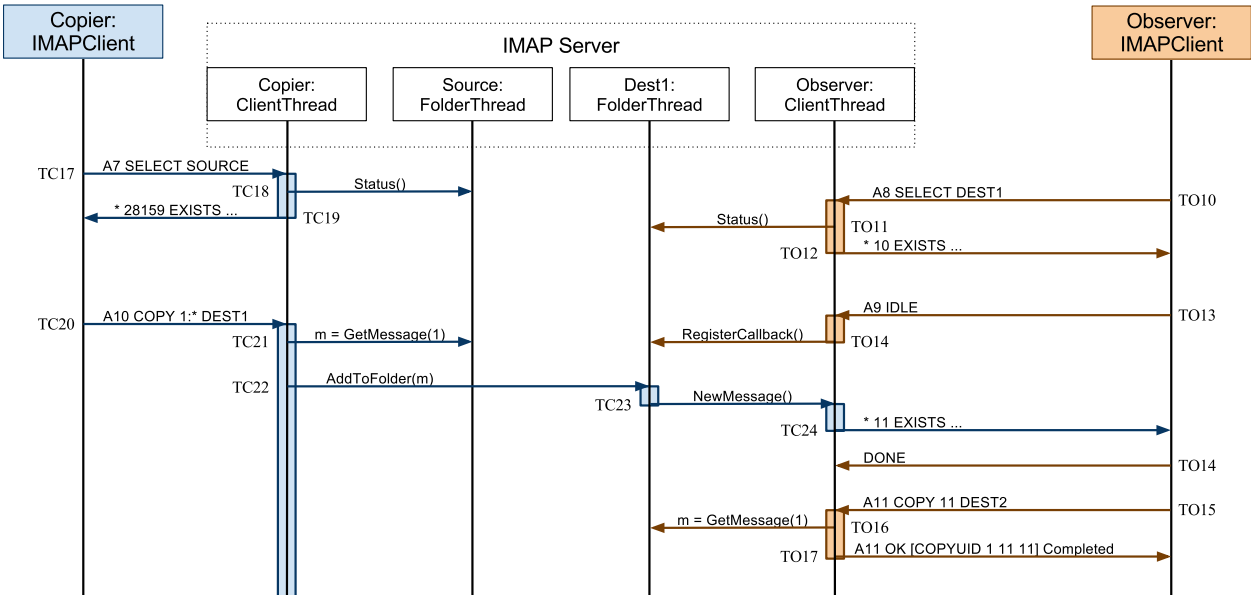


Figure 5.4: Rollback of completed requests. A11 must be able to roll back if A10 is canceled. As a result, copies of regions in A11 must be preserved after TO17, the point where A11 stops executing.

5.1.5 Freeing copies of regions: the process of finalizing requests

The copies of regions stored with a request may be needed even after a request completes. For example, consider the sequence of IMAP requests shown in Figure 5.4. In this sequence A11 becomes dependent on A10 at TO16. If A11 frees its copies of regions when it completes at TO17 there will be no way to roll back A11 if A10 is later canceled. Therefore, A11 must retain its copies of regions after it completes. Once A10 completes there is no need to retain A11's copies of regions. The process of determining this is called finalization.

Finalization is closely tied to undo(). Therefore, the design for the algorithm for finalizing shown in Listing 5.5, starts by simulating the first phase of undo(). The algorithm starts by doing a pseudo_undo() of every request that is not complete and not final. The pseudo_undo() is similar to the simple_undo() function, but does not change the content of regions and does not manipulate the generation number stored in regions. To search for dependencies, pseudo_undo() stores the earlier generation numbers that would have resulted from the rollback of regions in a map called pseudoTimes. It then checks all of the completed requests to see if the earlier generation numbers would have caused them to roll back.

When the checks terminate the requests in checked can be finalized, since the algorithm has established that it is impossible for dependencies to cause these operations to roll back. The runtime can free their checkpoints.

5.1.6 Implementing the stop() operation

Many of the patterns of redirection defined in Section 2.6 rely on the stop(request) operation, which pauses a request while saving as many of its changes as possible. A trivial, but potentially inefficient, approach to implementing the stop(request) is to call undo(request). Similarly, the continue(request) operation can be implemented in terms of start(request). These implementations ensure both that the stop() and continue() methods will be short and also that the system's state will be consistent. However, they discard all of the work done by the request before it is stopped.

```
void pseudo_undo(map<region_t*, generation_t> *pseudoTimes, request_t *req) {
    foreach ((region_t *region, writeEvent_t ev) in req.writeSet)
        if ((!region in pseudoTimes) || (pseudoTimes[region] > ev.time))
            pseudoTimes[region] = ev.time;
    }
}

void finalize {
    map<region_t*, generation_t> pseudoTimes;
    queue<request_t *> toCheck;
    queue<request_t *> checked;

    stop_the_world();
    foreach (request_t *request in allRequests) {
        if (request.state == finalized) {
            continue;
        } else if (request.state == completed) {
            toCheck.add(request);
        } else {
            pseudo_undo(&pseudoTimes, request);
        }
    }

    while (request = toCheck.removeFront()) {
        bool addToChecked = true;
        foreach ((region_t *region, generation_t t) in request.readSet) {
            if ((region in pseudoTimes) && (pseudoTimes[region] < t)) {
                pseudo_undo(&pseudoTimes, request);
                toCheck.add(checked);
                checked.clear();
                addToChecked = false;
                break;
            }
        }
        if (addToChecked)
            checked.add(request);
    }
    foreach (request in checked) {
        request.state = finalized;
        foreach ((region_t *region, writeEvent_t ev) in request.writeSet)
            free_checkpoint(ev.chkpt);
        foreach ((block_t *block) in request.deferredFree)
            free(block);
        request.readSet.clear();
        request.writeSet.clear();
        request.deferredFree.clear();
    }
    start_the_world();
}
```

Listing 5.5: Pseudo-code for finalization

Developers can refine the implementation of stop() and continue() to reduce the amount of work lost by using directives to mark atomic sections of code. These directives, introduced in Chapter 3, include:

- start_atomic(): The block of code up to the corresponding invocation of end_atomic() must execute to completion and must not be interrupted by the stop() operation. The block of code must execute within a short period of time. The restrictions on the undo() of atomic blocks have been defined to allow mediators to use atomic blocks to protect their data structures.
- atomic_sections_are_marked(): Developers have marked all of the atomic blocks in the code that is executing. If the runtime can establish that the callers can stop before this code completes, the runtime may safely stop any thread executing this code if it is outside of a marked atomic block.
- short_duration_lock(void *lock): The referenced lock will be held only for short period of time. Therefore, the runtime may be able to avoid an undo by waiting for the lock to be released.

The runtime uses these directives in the following algorithm:

1. Mark the request as stopping.
2. Any thread executing an associate_request() directive for a stopping request will stop(). As a result, no new threads can enter the set of threads associated with a stopping request. The threads in this set are called **associated threads**.
3. Any associated thread calling deassociate_request() will immediately be removed from the set of associated threads.
4. If an associated thread calls complete_request(), then the request will leave the stopping state and enter the completed state. The runtime will return a value from the stop() operation to indicate that the request completed rather than stopping. In some cases the system will need to report this event to the client, since it may change the interpretation of future requests.
5. If any associated thread is in an atomic block and holds a long lock the runtime will report inconsistent annotations: the thread engaged in a long duration operation within an atomic block.
6. If any associated thread is executing code where atomic sections are not marked, the runtime must eventually roll back the request with the undo() operation. This rollback may be delayed to allow other threads to exit atomic blocks.
7. The runtime will wait for threads to exit atomic blocks, stopping these threads when they exit the outermost block. The threads will all exit the blocks within a short time.
8. If a thread is executing code where atomic sections are marked and is not in an atomic block and holds no locks, the thread can safely stop.
9. The other associated threads will STOP, one at a time, as each thread exits all of its atomic blocks and releases all of its short-duration locks.
10. Once all of the associated threads have stopped, then the request enters the stopped state. Since developers ensure that all atomic blocks and short duration locks are short, a stopping request will reach the stopped state in a short period of time.

5.1.7 Tollgates

Many attentive systems incorporate third party modules that do not provide directives to the runtime. A *tollgate* is a wrapper that surrounds these modules, providing an approximation of the information that would normally be provided by directives within the module. The tollgate is constructed from modifiers that developers add to the module's interface. The runtime uses the tollgate to manage access to a region, called the tollgate region (tgr), that is shared by all of the third party modules. Using a single tollgate region for all of the modules allows the runtime to detect dependencies that arise from communication among the

Atomic sections	In atomic block?	Locks held	Action
Unmarked	Either	Any	Roll back
Marked	No	None	Stop
Marked	No	All short	Wait
Marked	No	Some long	Roll back
Marked	Yes	None or all short	Wait
Marked	Yes	Some long	Error: long duration atomic block

Table 5.1: Rules for stopping requests. Atomic block are honored even when atomic sections are not marked to simplify the implementation of redo logs. In this table the rules are sorted to demonstrate that all of the cases have been handled.

modules. The discussion below focuses on the tollgate from the perspective of propagating dependencies and managing changes in the tollgate region. For more details on the modifiers see Section 3.5.

During unchecked execution the runtime looks exclusively at the reader, writer, and independent modifiers on the interface. When a thread calls a function marked with reader, the runtime executes the enter_as_reader(tgr) operation before calling the function and executes exit_as_reader(tgr) when the function returns. The runtime makes similar calls to enter_as_writer(tgr) and exit_as_writer(tgr) when a thread calls a function marked with the writer modifier. The runtime takes no action for functions marked with the independent modifier.

The approach described above is very conservative, and reflects a practical limitation placed on developers. The information hiding principle [79] makes it impossible for developers to know the implementation details of some of their modules. Module developers can easily introduce sharing of state in ways that are not apparent at the module's interface by using of static fields, pointers in opaque structures, and global variables. Prior experience has indicated that this sharing can create unexpected dependencies among threads that access the module [59].

5.1.8 Thread issues

The runtime must address three concerns that occur in multi-threaded systems: reviving threads that exit during requests, changing the run state of threads during redirection, and interrupting threads in long system calls during redirection. These issues are discussed below.

5.1.8.1 Life-cycle issues

One or more of the threads may exit while they are associated with a request. When the exiting thread was also created by the request there is no risk to the consistency of the system. In the event of a rollback the system will automatically create a new thread, if needed, after the rollback. However, in cases where the thread existed before the request started the runtime must make some provision for reviving the thread. Creating a new thread will not work, since it will have a new thread identifier and the old thread identifier may be saved in one or more regions that survive rollback. In addition, the revived thread must reproduce the values saved in the original thread's local storage. Therefore, the runtime intercepts the thread's attempt to exit and postpones it until its associated request has finalized. If the request is rolled back, the runtime revives the thread by restoring its previous state and allows it to continue. The runtime must simulate the effects of thread exit in pthread_join() to preserve the interface of the pthreads library.

5.1.8.2 Waking threads

In multi-threaded applications, preserving the correct run state of threads—either sleeping or running—is extremely important. If a CANCEL request fails to wake threads that should still be running there is a possibility that the application could deadlock. Restoring the thread state to what it was when the checkpoint can lead to failures. Consider an example where a thread is blocked in a system call when a checkpoint is taken. After the checkpoint, the system call completes and the thread wakes. Then the user sends a CANCEL request that restores the checkpoint. The thread may never wake if the runtime puts it to sleep as part of rollback: the system call is no longer active in the kernel.

The behavior of the pthread_cond_wait() and pthread_cond_signal() calls is especially subtle.[3] If a thread is sleeping on a condition variable and wakes while another request is active, there are two possible causes:

- The pthread_cond_signal() was generated by a collaborating system, either directly via a shared memory window or indirectly by sending a message. As a result, the pthread_cond_signal() is still valid after redirection and execution should proceed.
- The pthread_cond_signal() could have been generated by the redirected request. Note that it is possible for this to happen via an external collaborator. In this case, the first thread should actually be asleep when the request is rolled back.

The runtime can address these problems by implementing a simple rule: any thread that started to run during the redirected request should be restarted after the rollback. This rule exploits a feature of POSIX [76] and Java [55] threading, which state that code must use a mutex to guard wait() statements, and must also double check the state of the system against spurious awakenings before proceeding allowing the thread to proceed.

For most other system calls the correct system state is not clear in the context of the system call. Therefore developers must provide mediators to restore the system's state after rollback. These mediators can control the running state of the threads engaged in the system call.

5.1.8.3 Interrupting long system calls during rollback

When a request is rolled back, it is possible that one or more of its threads will be engaged in a long system call. It may not be possible to wait for this system call to complete before completing the redirection. While this problem could be avoided if developers used asynchronous versions of system calls [100], the runtime implements a more general approach that uses pthread_kill() call[4] to regain control of the thread. Any partially completed state changes in the process's address space will be reversed by the copying scheme described above, while state changes associated with collaborating systems will be handled by their mediators. The runtime will restore the thread's local state, including its instruction pointer, register contents, stack, and thread-local storage from a mediator that wraps the system call.

5.2 Checked execution

Checked execution provides the same services as trusted execution while verifying the accuracy of the information provided by directives. Most directives can be verified with a small number of relatively inexpensive checks. For example, the directives that mark short sections—begin_short_section() and end_short_section()—can be implemented with a counter and a timestamp. The counter starts at 0 and increments every time

[3] The corresponding Java methods are wait() and notify().

[4] The pthread_kill() call does not destroy the thread. Instead, it sends an asynchronous signal to the thread that interrupts the system call. This terminology, while confusing, is consistent with the terminology adopted for inter-process signaling in POSIX.

that the runtime sees a begin_short_section(). When the counter moves from 0 to 1 the runtime updates the timestamp. When the runtime encounters an end_short_section(), if first verifies that the counter is greater than 0. If not, the runtime reports that the begin_short_section() and end_short_section() directives do not nest properly. Second, the runtime obtains a new timestamp and calculates the elapsed time since the counter was updated. If a long time has elapsed, the runtime reports that the short section took too long to execute.

However, the directives related to consistency are much more difficult to verify. These directives make assertions about the system's future behavior at the level of individual memory accesses. To check these directives, the runtime would need to examine each access to memory. Some dynamic checkers for concurrency errors, such as Eraser [86], Helgrind [89], and FlashLight [49], work at this level. However, these dynamic checkers have been designed with the assumption that information about concurrency policy is not available and must be inferred from the behavior of the system. The runtime is able to exploit the untrusted information provided in directives to achieve a much higher level of efficiency. It translates the information into permissions changes on the page table entries of the thread executing the code, causing the memory protection hardware in the processor to generate a fault if the thread makes an access that is inconsistent with the information provided in the directive. As a result, the runtime can check these directives while increasing the running time of the system by only a factor of 3.

We must resolve several problems to implement checked execution:

- The runtime must be able to manipulate access permissions for each thread independently. In traditional systems, every thread in a process shares a common set of access permissions. We accomplish this by allocating page tables for each thread in the system.
- The threads in the system must share a common view of memory. Normally, this happens automatically due to the sharing of access permissions among threads. However, this functionality is lost when we allocate thread-specific page tables and must be reimplemented by the runtime.
- The runtime must ensure that threads do not acquire conflicting permissions to access sound regions. While some policies, including guarded regions, ensure this, others, such as thread-confined regions, give developers the ability to control access to regions directly. These checks are sensitive to the interleaving among threads, but the runtime is able to ensure that there are no violations of the policy for the observed execution.
- Some systems use blocks that are too small to be represented in page tables, such as individual elements of arrays. The runtime must provide a reasonably efficient way to verify these blocks.
- Some systems use modules that do not have directives. While the runtime cannot check blocks of memory that are hidden in these modules, it should ensure that the directives applied to blocks outside the module are honored. In addition, it should ensure that the system does not access blocks supposedly hidden within the module.

The sections below discuss each of the issues mentioned above and describe some of the implementation details of the runtime system for checked execution. The section concludes by quantifying the performance of the runtime as it executes benchmarks chosen from the PARSEC benchmark suite.

5.2.1 The process model and filaments

POSIX threads share a common address space, permissions to access parts of this address space, and resources such as file descriptors. POSIX processes generally do not share address spaces and resources. However, it is possible to construct POSIX processes that share memory by employing the mmap() system call to map parts of a file into the memory of both processes. Each process is able to control its permissions to access this memory by manipulating the flags of the call.

The runtime with dynamic checking relies on an entity called a *filament* that shares features of processes

and threads. Like a process, the filament has control over its permissions to access memory. Like a thread, the filament shares resources such as files, network sockets, and a common address space.

We discovered that we could construct and control filaments without modifying the pthreads library used on Linux systems by intercepting the clone() system call generated by pthread_create(). The clone() system call creates either a new thread or a new process depending on a set of flags that are passed as one of its parameters. In general there is a flag for each piece of state that could be shared. We create new filaments by clearing three of the flags: CLONE_VM, CLONE_THREAD, and CLONE_SIGHAND.

Clearing CLONE_VM causes the kernel to allocate a new set of page tables for the filament. This allows us to manipulate the permission bits in the filament's page tables to ensure that its accesses do not violate the policies specified in directives.

Clearing CLONE_THREAD allows the new thread to call exec() without terminating the process. We use this functionality to run the debugger when the dynamic checker detects an inconsistent directive.

Finally, we clear CLONE_SIGHAND because the clone() system call will fail if CLONE_SIGHAND is set while CLONE_VM is not set. By clearing this flag we are responsible for propagating changes in signal handlers among the filaments of the system. We have not yet implemented this feature because the systems that we have examined do not make use of signal handlers. We would probably propagate this information through the log described below.

It is important to note that filaments are not totally independent processes. For example, filaments share a common set of file descriptors just as threads do. As a result, we do not have to change the implementation of system calls that work with file descriptors. This is the principal advantage to using the clone() system call rather than creating a separate process with fork(). However, since the clone() call is Linux-specific it is possible that later kernels may change the call in ways that make it impossible to create filaments.

5.2.2 Propagating changes among filaments

The filaments provided by the clone() system call are not ideal. Since the page tables for each filament are unique, memory allocation events are not automatically propagated among filaments. This in turn, points to an assumption built into the design of most processors that uses a single data structure to control both the content of the address space and also the permissions to access the address space.

We overcome this limitation by using a log to inform filaments of the creation of new blocks of memory. This log consists of an array of pointers to the data structures that describe regions. These data structures are preallocated in a block of memory, and both the log and the block memory containing the data structures are automatically mapped in to new filaments when they are created.

Filaments discover new regions asynchronously. When a filament is ready to create a new region it locks the log, updates its page tables with any new regions that it finds in the log, allocates the region, and adds a pointer to the region to the log, and then releases the lock. The lock ensures that filaments do not inadvertently allocate overlapping regions. We also check the log when acquiring locks, creating new threads, and processing the get_transferable() and release_transferable() directives. By checking the log on these occasions, we ensure that we discover new regions using a sound policy before accessing them.

However, it is possible for a filament to access a region with an unsound access policy before becoming aware of the existence of the region. The filament will not have access to the memory associated with the region, causing the access to generate a fault. We check for this case in the error reporting code, which handles it by processing any outstanding log entries and restarting the filament.

5.2.3 Checking transfer directives

Developers place the following directives in code to control the assignment of permissions to filaments for both thread-confined regions and phased immutable regions:

- get_transferable()
- get_transferable_ro()
- release_transferable()
- release_transferable_ro()

If the system does not coordinate these directives, for example by using locks, filaments could attempt to obtain conflicting permissions to access the region. The runtime will report attempts to gain conflicting permissions as fatal errors.

The current implementation of checked execution ensures that no two filaments gain conflicting permissions by examining the region's data structure when it encounters these directives. It generates an error if a filament attempts to gain write access while another filament has access to the region and when a filament attempts to gain read access while a writer is active. We call this approach **best effort checking**.

However, it is possible for developers to write poorly synchronized directives that will occasionally slip past a dynamic checker relying on best effort checking. An example is given in Listing 5.6. Some interleavings will allow this program to run to completion without creating a data race. Other interleavings could cause the program to read an uninitialized pointer at line B35, potentially creating bad output or a crash. Finally, it is possible that the get_transferable() annotation at line B5 and the get_transferable_ro() annotation at line B33 could overlap, causing the dynamic checker to report a transfer race on either the parent or the child thread.

This lack of determinism happens because the transfer directives are not sufficiently synchronized in the program due to the missing pthread_join() call at line B31. Ideally we would like the dynamic checker to detect that the transfers are not sufficiently synchronized and report an error. However, this problem is not easily solved, since there are many possible techniques that can be used to synchronize the transfers. The variant of the example shown in Listing 5.7 relies on the happens-before relationships created by pthread_create() and pthread_join() to synchronize the permissions changes. This approach is often used when a parent thread spawns one or more worker threads.

It is also possible to synchronize the transfers by using a combination of condition variables and synchronized blocks. This approach is shown in Listing 5.8. Finally, it is possible to synchronize transfers by passing a message via either an in-memory queue or network socket to indicate that it is safe to transfer. An example of this form of synchronization is given in Listing 5.9. In this example the read() at line M35 establishes a happens-before relationship between the child's release_transferable() at M14 and the parent's get_transferable_ro() at M37. This is only the case because socket[0] and socket[1] are connected by the socketpair() call. Direct communication between threads via sockets is less efficient than other forms of communication. However, in cases where threads use sockets to communicate with an outside system, such as an IMAP server, this sort of synchronization is possible.

There are several strategies that we could employ to improve best-effort checking: missing happens-before detection, annotated happens-before checking, and full behavior modeling. Each of these approaches involves trade-offs among annotation effort, false-positives, false-negatives, the types of systems that can be annotated, and the complexity of analysis. We will discuss these techniques in more detail below.

Missing happens-before detection involves tracking the happens-before relationships created among filaments by thread creation, thread joins, and standard concurrency control constructs. The detection will create an error when a filament attempts to obtain permissions that were released by a different filament and no-happens before relationship was established. This approach does catch some errors missed by best-effort checking and adds no annotation effort. However, it suffers both from false positives and false negatives and

```
B1   typedef struct {int i; char *o; } job;

B3   void *do_work(void *ctx) {
B4     job *j = ctx;
B5     get_transferable(j);
B6     region_t saved = bind(region_of(j));
B7     j->o = malloc(10);
B8     bind(saved);

B10    snprintf(j->o, 10, "%d", j->i);

B12    release_transferable(tW);

B14    return NULL;
B15  }

B17  int main(int argc, char *argv[]) {
B18    pthread_t c;

B20    region_t saved = bind(new_region(PHASED_IMMUTABLE));
B21    job *j = malloc(sizeof(*j));
B22    bind(saved);

B24    j->i = atoi(argv[1]);

B26    release_transferable(j);

B28    pthread_create(&c, NULL, do_work, j);
B29    ...
B30    /* Removing the line below creates a transfer race */
B31    /* pthread_join(c, NULL); */

B33    get_transferable_ro(j);

B35    printf("%s\n", j->o);

B37    release_transferable_ro(j);

B39    return 0;
B40  }
```

Line	Valid	Justification
B26	Y	
B5	Y	pthread_create() at B28
B12	Y	same thread
B33	N	Last common event was B26, needed B12

or

Line	Valid	Justification
B26	Y	
B33	Y	same thread
B37	Y	same thread
B5	N	Last common event was B26, needed B37

Table 5.2: These tables show two executions of the code to the left. In both tables a directive executes without establishing a happens-before relationship. A 'Y' in the valid column indicates that the happens-before relationship was established and the relationship is described in the Justification column.

Listing 5.6: An example of a transfer race. The missing pthread_join() call at line B31 could cause a transfer race depending on the relative progress of threads. If the runtime checks for happens-before relationships this race will always be caught. Table 5.2, above and to the right, show the two possible executions.

```
A1   typedef struct {int i; char *o; } job;

A3   void *do_work(void *ctx) {
A4     job *j = ctx;
A5     get_transferable(j);
A6     region_t saved = bind(region_of(j));
A7     j->o = malloc(10);
A8     bind(saved);

A10    snprintf(j->o, 10, "%d", j->i);

A12    release_transferable(tW);

A14    return NULL;
A15  }

A17  int main(int argc, char *argv[]) {
A18    pthread_t c;

A20    region_t saved = bind(new_region(PHASED_IMMUTABLE));
A21    job *j = malloc(sizeof(*j));
A22    bind(saved);

A24    j->i = atoi(argv[1]);

A26    release_transferable(j);

A28    pthread_create(&c, NULL, do_work, j);
A29    ...
A30    pthread_join(c, NULL);

A32    get_transferable_ro(j);

A34    printf("%s\n", j->o);

A36    release_transferable_ro(j);

A38    return 0;
A39  }
```

Line	Valid	Justification
A26	Y	
A5	Y	pthread_create() at A28
A12	Y	same thread
A30	Y	pthread_join() is always valid
A32	Y	pthread_join() implies A12 *HB* A30

Table 5.3: This table shows an execution of the code to the left. The code always establishes the necessary happens-before (*HB*) relationship before reaching a directive. A 'Y' in the valid column indicates that the happens-before relationship was established and the relationship is described in the Justification column.

Listing 5.7: Directives to describe a thread-confined job block, allocated on line A21. The directives in this listing are shown in bold type.

```
C1  typedef struct {
C2      int i;
C3      char *o;
C4      int bDone;
C5      pthread_mutex_t lock;
C6      pthread_cond_t done;
C7  } job;

C9  void *do_work(void *ctx) {
C10   job *j = ctx;
C11   get_transferable(j);
C12   region_t saved = bind(region_of(j));
C13   j->o = malloc(10);
C14   bind(saved);

C16   snprintf(j->o, 10, "%d", j->i);

C18   release_transferable(tW);

C20   pthread_mutex_lock(&j->lock);
C21   j->bDone = 1;
C22   pthread_cond_signal(&j->done);
C23   pthread_mutex_unlock(&j->lock);

C25   return NULL;
C26 }

C28 int main(int argc, char *argv[]) {
C29   pthread_t c;

C31   region_t saved = bind(new_region(PHASED_IMMUTABLE));
C32   job *j = malloc(sizeof(*j));
C33   bind(saved);
C34   j->bDone = 0;
C35   pthread_mutex_init(&j->lock, NULL);
C36   pthread_cond_init(&j->done, NULL);

C38   j->i = atoi(argv[1]);

C40   release_transferable(j);

C42   pthread_create(&c, NULL, do_work, j);

C44   pthread_mutex_lock(&j->lock);
C45   while (!j->bDone) {
C46     pthread_cond_wait(&j->done, &j->lock);
C47   }
C48   pthread_mutex_unlock(&j->lock);

C50   get_transferable_ro(j);

C52   printf("%s\n", j->o);

C54   release_transferable_ro(j);
C55   return 0;
C56 }
```

Listing 5.8: An example of transfers synchronized via locking and condition variables.

Line	Valid	Justification
C40	Y	
C44	Y	pthread_mutex_lock() is always valid
C46a	Y	j->lock acquired at C44
C11	Y	pthread_create() at C42
C18	Y	Same thread
C20	Y	pthread_mutex_lock() is always valid
C21-C23	Y	j->lock is held from C20
C46a	Y	Acquires &j->lock
C48	Y	j->lock was acquired at C46a
C50	Y	C18 *HB* C50 established by C46a
C54	Y	same thread

Table 5.4: Fully synchronized transfers will be accepted by missing happens-before detection. * Many other interleavings are also possible.

```
M1  typedef struct {int i; char *o;} job;

M3  int sockets[2];

M5  void *do_work(void *ctx) {
M6    job *j = ctx;
M7    get_transferable(j);
M8    region_t saved = bind(region_of(j));
M9    j->o = malloc(10);
M10   bind(saved);

M12   snprintf(j->o, 10, "%d", j->i);

M14   release_transferable(tW);

M16   write(socket[1], &j, sizeof(j));

M18   return NULL;
M19 }

M21 int main(int argc, char *argv[]) {
M22   pthread_t c;

M24   socketpair(..., sockets);
M25   region_t saved = bind(new_region(PHASED_IMMUTABLE));
M26   job *j = malloc(sizeof(*j));
M27   bind(saved);

M29   j->i = atoi(argv[1]);

M31   release_transferable(j);

M33   pthread_create(&c, NULL, do_work, j);

M35   read(socket[0], &j, sizeof(j));

M37   get_transferable_ro(j);

M39   printf("%s\n", j->o);

M41   release_transferable_ro(j);

M43   return 0;
M44 }
```

Line	Valid	Justification
M31	Y	
M7	Y	release_transferable() at M31 before M37 before M7
M14	Y	same thread as M7
M37	Y	release_transferable() at M14 before write() at M16 before read at M35
M41	Y	same thread at M37

Listing 5.9: An example of transfers synchronized by communication through a socket.

also cannot be applied to some systems.

This approach is sufficient to detect the potential for transfer races in Listing 5.6. This can be seen by examining the four possible interleavings of statements B26, B5, B12, B33, and B37.

Two of the interleavings, B26:B5:B33 and B26:B33:B5, are caught even by best-effort checking and will not be considered further in this section. The two remaining interleavings are B26:B5:B12:B33:B37 and B26:B33:B37:B5:B12. Both of these interleavings are free of data races. However, the B26:B33... variant involves a reference through an uninitialized pointer at B35. The analysis shown in Table 5.2 demonstrates that missing happens-before detection will reject both of these interleavings as transfer races due to the lack of synchronization caused by the missing pthread_join() at B31. The version of this code shown in Listing 5.7 also limits executions to interleavings that are true negatives under missing happens-before analysis. Examples of true negatives generated by missing happens-before analysis for these cases are shown in Table 5.4.

Unfortunately, missing happens-before analysis is subject to both false positives and false negatives. The code shown in Listing 5.9 will lead to false positives because the read() and write() system calls normally only act as memory barriers for the threads that issue them. A happens-before relationship can be established only by examining the state of the sockets to determine that they are connected and that there is no data pending on socket[0] before the write call at M16. There are many calls that could add and remove data from sockets, and applications are also free to create communication protocols that are much more complex than the simple example given here. Therefore, it is not trivial for a dynamic analysis to infer these relationships.

False negatives occur when missing happens-before creates happens-before relationships based on synchronization that does not actually protect the transfers. This error can be quite subtle, as can be seen in Listing 5.10. The dynamic checker may accept the interleaving shown in Table 5.5 due to the creation of a false, from the perspective of the transfer, happens-before relationship created by the locking inside puts(), which is called at FP13 and FP34.

Directives for happens-before checking would eliminate the false negatives of missing happens-before analysis by requiring developers to justify the safety of get_transferable() and get_transferable_ro() by referring to specific happens-before relationships in their code. These directives would be limited, allowing developers to specify only transfer policies that could be verified by dynamic checking. These directives may not be able to handle all sound synchronization policies, and developers may find them to be difficult to apply. For example, systems like the one shown in Listing 5.9 would not fit into any simple system of directives.

Full behavior modeling would be needed to handle cases like the one shown in Listing 5.9. The modeling would be complex, reasoning about system states, the contents of various variables, and isolation guarantees. In the example given in this listing, correct synchronization depends on the following properties:

- socket[0] and socket[1] are connected by socketpair()
- The identifiers in socket[0] and socket[1] remain unchanged
- socket[0] and socket[1] remain connected
- socket[0] has no queued data
- There are no other writes to socket[1]

These properties are preserved in Listing 5.9, but could be difficult to verify in more complex systems. The resulting system of directives would be very complex and would probably constitute a simple model of the system, necessitating a model checking approach to establish the soundness of the synchronization policy. This approach would be similar to the one that Microsoft adopted for SLAM [9].

```
FP1   typedef struct {int i; char *o; } job;

FP3   void *do_work(void *ctx) {
FP4     job *j = ctx;
FP5     get_transferable(j);
FP6     region_t saved = bind(region_of(j));
FP7     j->o = malloc(10);
FP8     bind(saved);

FP10    snprintf(j->o, 10, "%d", j->i);

FP12    release_transferable(tW);
FP13    puts("Child done\n");

FP15    return NULL;
FP16  }

FP18  int main(int argc, char *argv[]) {
FP19    pthread_t c;

FP21    region_t saved = bind(new_region(PHASED_IMMUTABLE));
FP22    job *j = malloc(sizeof(*j));
FP23    bind(saved);

FP25    j->i = atoi(argv[1]);

FP27    release_transferable(j);

FP29    pthread_create(&c, NULL, do_work, j);
FP30    ...
FP31    /* Removing the line below creates a transfer race */
FP32    /* pthread_join(c, NULL); */

FP34    puts("Our results\n");

FP36    get_transferable_ro(j);

FP38    printf("%s\n", j->o);

FP40    release_transferable_ro(j);

FP42    return 0;
FP43  }
```

Listing 5.10: Some transfer races will escape missing happens-before checking. The C runtime library's implementation of puts() is thread-safe. Therefore, it is possible to define a happens-before order between the puts() calls at FP13 and FP34. The dynamic checker may detect that a happens-before relationship has been established among the filaments. However, the presence of the happens-before relationship does not indicate that there is sufficient synchronization to ensure that FP12 will always happen before FP36. Therefore, under some interleavings the runtime will falsely infer that the transfers are safe. In other interleavings the runtime will detect a transfer race.

Line	Valid	Justification
FP27	Y	
FP5	Y	pthread_create() at FP29
FP12	Y	Same thread
FP13	Y	puts(): FP12 *HB* FP13
FP34	Y	puts(): FP13 *HB* FP34
FP36	False negative	Same thread: FP34 *HB* FP36 *HB* is transitive: FP12 *HB* FP34
FP40	Y	Same thread: FP 36 *HB* FP40

Table 5.5: Example of a false negative via puts() when using happens-before (*HB*) relationships to validate permission changes. In this interleaving puts() created a happens-before relationship between the parent and child threads even though the threads were not synchronized. An interleaving that swaps FP13 and FP34 is possible and would lead to an error under the same analysis.

5.2.4 Checking array slices

The prior discussion has assumed that blocks are individual blocks of memory on the heap. Some systems share state in arrays, attaching policies to ranges of array indexes. To accommodate this, the runtime uses a new data structure called a sliceable array.

In most cases the interface to sliceable arrays hides the internal representation. There are at least three ways to check accesses to sliceable arrays. Infrequently accessed arrays made up of small blocks with a small transfer granularity can be checked efficiently by assigning a region to each index and checking accesses in software. The current implementation uses C macros to implement this approach.

If the granularity is predictable and allows the slices to be mapped to unique pages the runtime can manage the slices as blocks in a region. In some cases it may be necessary to add padding to the array to accomplish this. This approach avoids much of the overhead of checking array accesses in software. In addition, there is no need to rewrite the software to replace the array reference operators with macros. Finally, it allows us to expose pointers to the array to modules without directives, including assembly routines optimized to use SIMD instructions, without losing the ability to check the policy attached to the array. This functionality was needed in some of the benchmarks discussed below.

5.2.5 Tollgates

A tollgate is a boundary between a system using directives and a module that does not have directives. The tollgate scopes the dynamic checks in the system, ensuring that directives continue to apply to blocks controlled by the system while ensuring that accesses to blocks owned by the module do not produce errors. The tollgate also controls the transfer of blocks between the module and the system.

The behavior of the tollgate is defined by modifiers that are attached to function signatures, as described in Section 3.5.1. This section describes the approach that the dynamic checker uses to check the policies specified by these modifiers. First, it discusses the implementation of tollgates and the way that permissions are managed to ensure that unchecked blocks do not leak through tollgates. Next, it describes the behavior of the runtime as it attaches a policy to a previously unchecked block as a thread passes through the tollgate. Then it considers the behavior of the dynamic checker when new blocks are created within a module without directives. Finally it describes the behavior of the checker when a checked block is converted to an unchecked block.

5.2.5.1 Tollgate implementation

In the runtime tollgates are implemented as wrapper functions. The runtime reroutes calls from the system to the module through the tollgate wrapper by using a #define preprocessor directive to change the function's name when it is called. This approach allows us to avoid intercepting calls that the wrapped module makes to itself and also avoid calls between the modules protected by tollgates.

The runtime uses a special region, called the tollgate region, to hold all of the unchecked blocks in the system. It does not attempt to associate blocks with specific modules, since the modules behind a tollgate may share blocks in ways that cannot be predicted from their interfaces. It ensures that the code in the system does not gain access to blocks controlled by these modules by issuing the mprotect() system call to disable access permissions to the tollgate region when control returns to the system through the tollgate. By default, every block allocated while code is executing within the tollgate is placed in the tollgate region.

5.2.5.2 Checking the reader, writer, and independent directives

Many modules contain functions that do not access blocks within the tollgate region. For example, most implementations of strcmp(str1, str2) function access only the two blocks passed to the function. The overhead of the mprotect() calls could easily be greater than the processing time of these functions. The runtime relies on the following directives, which developers add to the function signatures that define the module's interface:

- Independent indicates that the function will not access blocks in the tollgate region. Therefore, mprotect() calls are unnecessary when the function enters and exits the tollgate. Any attempt to access a block within the module will generate a fault, causing the runtime to report that the independent directive is inaccurate. This is a fatal error, since the directive also controls the propagation of dependencies among requests.
- Reader indicates that the function will read, but not modify, blocks within the tollgate region. The tollgate should grant read, but not write, permission to the blocks in the tollgate region. To do this the tollgate will have to issue mprotect() calls both when control enters the module and also when control returns from the module. There is no performance benefit to the reader directive, but checked execution should verify it because it affects the dependencies created when requests enter the tollgate.
- Writer indicates that the function will read and modify blocks in the tollgate. The mprotect() calls are needed. The dynamic checker may attempt to identify places where the writer directive was unnecessary to improve the quality of the tollgate. It will do this by postponing the mprotect() call when control passes through the tollgate, issuing it only if a fault shows that the module attempted to access one of the blocks in the tollgate region. This both improves performance and provides advice to developers that can be used to make the tollgate more accurate.

5.2.5.3 Placing newly allocated blocks into the tollgate region

Functions like strdup() create a new block but return control of the block to the system rather than retaining a reference to it. In abstract terms, new blocks created by a module behind a tollgate are immediately placed in the tollgate region. However, recovering access to these blocks is quite expensive. In addition, placing these blocks into the tollgate region would force developers to apply the writer modifier to these functions.

Therefore, blocks allocated within tollgates are initially placed in a region following the serial thread confinement policy. The function executing within the module retains exclusive access to these blocks until one of two events occurs: the function returns through the tollgate or another thread, executing code behind the tollgate, attempts to access the block.

When control returns through the tollgate, the tollgate first processes the modifiers that allow the caller to claim blocks. Blocks are removed from the thread-confined region as they are claimed. There is no need to retrieve exclusive access to these blocks before claiming them: their presence in the thread's region indicates that no other thread has gained access to the blocks. Any blocks that remain in the region after the modifiers have been processed are moved to the tollgate region.

If another filament attempts to access one of the newly allocated blocks before it is placed in the tollgate region, the processor will generate a fault. The runtime will immediately suspend the allocating filament. It will move the block into the tollgate region, removing it from the allocator's thread-confined region, and then restart both the allocating thread and the thread attempting to access the block.

5.2.5.4 Removing blocks from the tollgate region

Callers can gain exclusive access to blocks in the tollgate region as a result of executing a function in the tollgate. To soundly assign access to the caller, the tollgate must remove the block from the tollgate region,

suspend any other threads executing in modules protected by tollgates, and remove their permissions to access the block before reassigning the block to the caller. This procedure involves a TLB shoot-down, which could involve substantial delays on the thread exiting the tollgate. However, it must be completed before allowing execution of the thread to continue, since it prevents other threads that are executing in the tollgate from accessing the block after it is returned to the caller. If the caller assumes that the block is protected by a sound access policy, these accesses could create undetected data races and dependencies. The current runtime does not implement this feature.

5.2.5.5 Adding the system's blocks to the tollgate region

Some functions, including free(), transfer control of a block from the system to a module protected by a tollgate. The procedure transferring the block is relatively simple. First the tollgate ensures that the caller has established exclusive access to the block. Next, the tollgate ensures that the policy for the block allows it to be transferred. There are two policies which must be considered. If the block was allocated with the thread-local policy it cannot placed into the tollgate region, since doing so would imply that the block could be accessed by another filament. In addition, blocks that have been previously published with the immutable policy cannot be converted, since this implies that the block could be modified at some point in the future. All of the other policies listed in Section 3.3.2 support conversion.

Once the tollgate has determined that the conversion is valid, it removes the block from its current region and places the block into the tollgate region. The tollgate then proceeds to change the permissions to the HIDDEN region, as described in Section 5.2.5.1.

5.2.6 Implementation details

There are two aspects of the runtime that may be modified in future versions: the approach to error reporting and the approach to handling the termination of filaments. It is unlikely that these design choices affect the performance numbers reported below.

5.2.6.1 Reporting errors

The runtime installs a signal handler to intercept the segmentation violation signal (SIGSEGV) that is generated when a filament attempts to access memory without first obtaining permission to do so. In some situations the signal handler may be called even though no policy has been violated. Therefore, the signal handler confirms that there has been a policy violation by finding the region that corresponds to the address that generated the signal. It consults the region to see if the current filament should have access to the memory. If the signal is a true access violation the handler reports the error by either entering GDB, a debugger, or by writing a log file and terminating the process.

False faults are often caused by regions in the log that have not been mapped into the filament's page tables. In these cases the runtime processes the new log entries and returns from the signal handler, allowing execution to continue. False faults are expensive, often costing roughly 6,000 cycles[5] of lost execution time. Therefore the runtime attempts to avoid false faults by checking for unprocessed log entries around the calls that establish happens-before relationships among filaments. In theory it could also avoid false faults by interrupting filaments whenever a new region is created. However, the performance impact of these interrupts would be similar to the impact of a false fault, and would happen every time a new region is created. In the current implementation false faults are extremely rare. It is likely that interrupting filaments would result in a less efficient implementation.

[5] This number was taken on an Intel® Core®2 Duo E7300 CPU running at 2.66 GHz

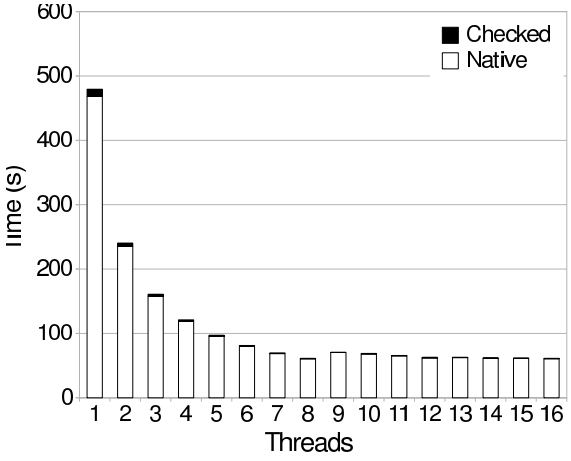


Figure 5.5: Time to complete BLACKSCHOLES native tests

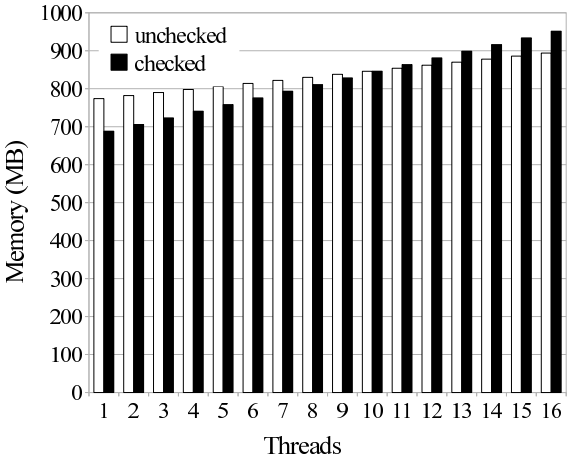


Figure 5.6: Memory use, BLACKSCHOLES native test

It is important to note that the runtime uses log entries only to grant new permissions to filaments, not to take existing permissions away. Filaments always relinquish permissions to memory voluntarily: either by calling release_transferable() for serial-thread-confined blocks, calling pthread_mutex_unlock() for guarded blocks, or exiting for thread-local blocks.

5.2.6.2 Kernel support

To date we have needed to make only one small modification to the kernel to support filaments. Normally, kernels check the number of threads in a given memory space when a thread is exiting. Kernels normally skip some of the processing needed to support pthread_join() when the last thread in a process exits. This optimization breaks pthread_join(), and so the runtime disables it. It could avoid this modification by starting and suspending a place-holder POSIX thread within each filament. However, doing so would complicate the runtime. In addition, we anticipate adding kernel support for the finalization of lazy checkpoints.

5.2.7 Overhead of checked execution

The evaluation of the runtime system used BLACKSCHOLES, SWAPTIONS, and X264, three programs from the PARSEC 2.0 benchmark suite. The results reported below are from the native series of tests, the largest tests in the benchmark suite. The times reported are from a system with two Intel® Xeon® CPUs (E5405) running at 2.00GHz under a modified 2.6.26-2-686 Debian kernel. The system has 4GB of RAM. The tests use 1 to 16 threads to execute the benchmarks. By exceeding the number of available cores, the tests will reveal overhead added by the filaments to the task switch time.Figure 5.9 breaks out time that is spent in the kernel in the mprotect() call. This time should be counted as part of the overhead for the dynamic checking.

BLACKSCHOLES

The BLACKSCHOLES benchmark places relatively few demands on the runtime. A master thread initializes several large structures containing data for a basket of options. It then starts a small number of long-running threads to price the options. These threads write their results into shared arrays before exiting. The modifications to the benchmark represent this as a sliceable array using software checking.

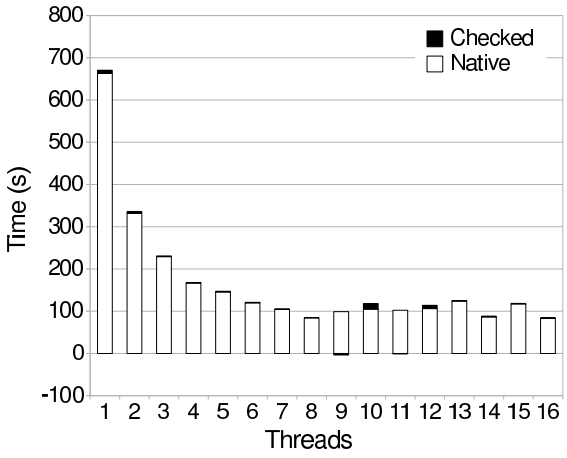


Figure 5.7: Time to complete SWAPTIONS native tests

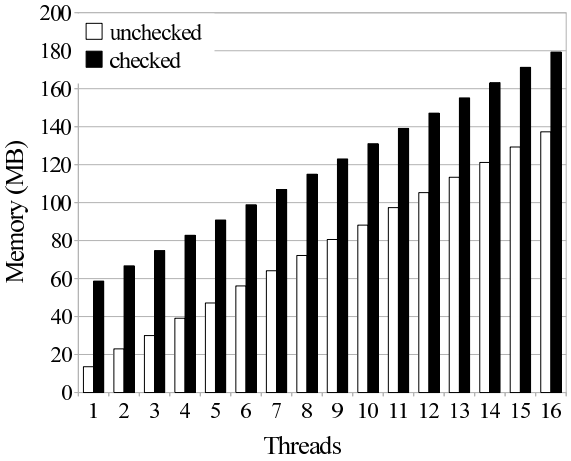


Figure 5.8: Memory use, SWAPTIONS native test

Figure 5.5 shows the results from the tests. The small number of transfers means that the dynamic checker adds very little overhead. The worst case overhead happens with two threads. The ratio of the execution time for the fully checked version of BLACKSCHOLES to the original threads version ranges from 1.00 to 1.02, with the worst case occurring at two threads. When the annotated version of the code is recompiled for use with pthreads the running time ratio varies between 0.99 and 1.01.

Memory is rarely allocated in the BLACKSCHOLES benchmark. Therefore, the lack of memory recycling in the runtime has little effect on the memory consumption, as shown in Figure 5.6. In fact, the results are occasionally lower because we removed unnecessary string duplications when the benchmark started.

SWAPTIONS

There is also relatively little overhead for enforcing policies in the SWAPTIONS benchmark. Unlike the other graphs in this section, Figure 5.7 computes the overhead of checking the directives by comparing the fully checked code to the annotated code compiled for traditional threading. We did this because the memory recycling modifications reduced the running time of the benchmark, at times exceeding the overhead added by the dynamic checker.

Figure 5.8 shows the memory consumption of SWAPTIONS after we implemented memory recycling in the application. The memory recycling holds the overhead for dynamic checking to a constant factor relative to the number of threads.

X264

As mentioned in Section 3.8, the tests with X264 use a more recent version of the program than the one included in the PARSEC benchmark suite. The later version has similar performance characteristics to the PARSEC version but makes more consistent use of locking.

The X264 benchmark is challenging, from both a modeling and a resource consumption point of view. A master thread spawns a series of worker threads, roughly one for each frame in the file being encoded. It hands each worker a frame to be encoded and references to the prior frames in the stream. Each worker frequently acquires locks and initiates small transfers of pages to update the other threads with its progress.

Figure 5.9 shows the performance of X264, comparing the execution time of the program when using

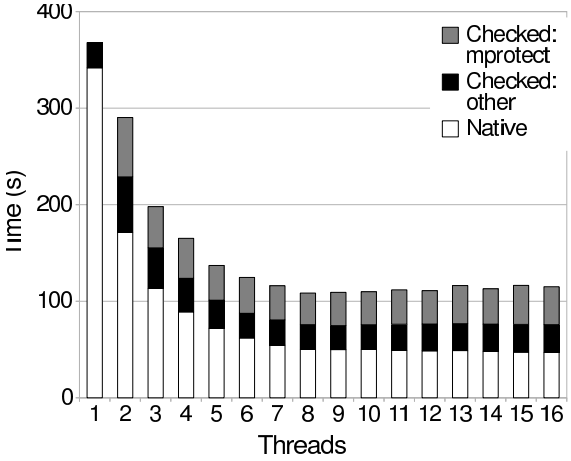


Figure 5.9: Time to complete x264 native tests

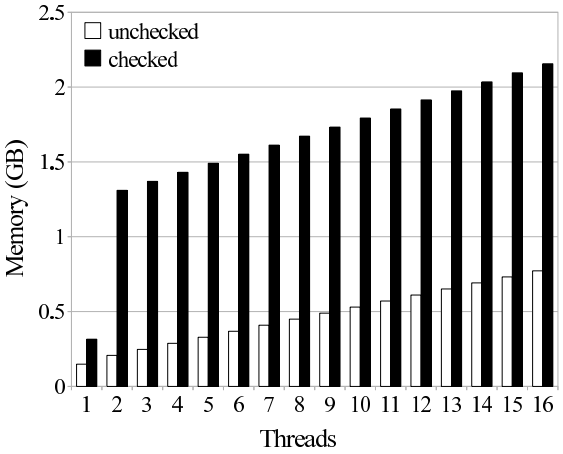


Figure 5.10: Memory use, x264 native tests

threads to the execution time using filaments. The large number of transfers causes a very large number of small mprotect() system calls, resulting in substantial kernel overhead. The overhead for the single-threaded case is modest because no transfers are initiated. The overhead appears to gradually increase with the number of threads. At 15 and 16 threads the execution time of the fully instrumented system is roughly 2.46 the time of the original version when executed by threads. When the modified code is compiled for execution with standard threads the running time is roughly comparable to the unmodified code, with an execution time of 1.03.

Figure 5.10 shows the memory overhead of the dynamic checking when running the x264 benchmark. In future work we plan to address this problem by implementing memory recycling in the dynamic checker.

Comparison to Helgrind

Due to the use of page tables, the runtime is much more efficient than other dynamic checkers that target data races. Table 5.6 shows a comparison of the overhead of the approach described here to the overhead of Helgrind 3.4.1. This table was constructed by timing Helgrind as it checked for data races in the "simlarge" data set using the unmodified code. The table expresses the overhead as a ratio, dividing the time to run the benchmark with the checker by the time to run the benchmark using native threads. A ratio of 1.0 indicates that the dynamic checker adds no detectable overhead to the execution time of the code.

The table shows that the overheads for filaments are much lower than the overheads for Helgrind. The difference in overhead can largely be attributed to the advance knowledge of the policy used to protect each block that the runtime obtains from the directives. This knowledge allows the runtime for filaments to target its data collection, while Helgrind must collect data about every shared block in the system. In addition, Helgrind's overhead increases as the number of threads increases, reflecting the fact that Helgrind does not allow threads to execute concurrently. Since the runtime allows threads to execute concurrently the overheads do not rise as quickly as the number of threads increases.

The directives also allow us to avoid generating false reports of data races. Helgrind reports 1-9 races for the BLACKSCHOLES benchmark, 1 data race for SWAPTIONS, and 50,000 - 219,000 data races for x264. Since we are able to apply sound access policies to every region in x264, we can conclude that all of these race reports are false.

	BLACKSCHOLES		SWAPTIONS		X264	
	Fil.	Hel.	Fil.	Hel.	Fil.	Hel.
1	1.0	31	1.0	50	1.1	155
2	1.1	68	1.0	104	1.8	484
3	1.2	100	1.1	151	2.0	731
4	1.0	120	1.1	206	2.2	947
5	1.0	143	1.1	205	2.3	1115
6	1.1	175	1.1	227	2.5	1254
7	1.1	180	1.0	307	2.3	1260
8	1.2	205	1.1	384	2.4	1318
9	1.1	159	1.1	353	2.4	1361
10	1.1	175	1.0	253	2.5	1356
11	1.1	180	1.2	228	2.5	1377
12	1.1	182	1.1	266	2.6	1366
13	1.1	190	1.2	176	2.7	1428
14	1.1	197	1.1	219	2.7	1435
15	1.2	197	1.1	289	2.6	1445
16	1.2	209	1.1	369	2.5	1366

Table 5.6: Overhead of filaments and Helgrind

5.3 Future work

The runtime system described in this chapter is a work in progress, and is not fully implemented. Microbenchmarks, along with the prototype runtime support discussed in Section 4.2, suggest that the runtime would be feasible. Implementing the runtime and applying it to a set of representative applications would both validate the design and also allow the performance impact of the runtime system to be quantified.

5.3.1 Comparing error rates

Other research projects are reducing the number of false races reported by Helgrind. Notably, Helgrind+ [61], an improved version of Helgrind, has eliminated the false reports for BLACKSCHOLES and SWAPTIONS, and has greatly reduced the error rate for X264.

However, the version of X264 that was annotated in this work does not correspond to the version used for the published results for these tools. In addition, we may have eliminated data races when we changed the reporting of scanlines completed to align transfers on page boundaries. We would like to directly compare the results generated by the runtime to the ones generated by these tools.

5.3.2 Assisted development of directives

It is often difficult to infer the design intent of a concurrent system by examining its code. However, traditional dynamic checkers routinely build models that are closely related to the directives. For example, the LockSet algorithm [86] constructs a model that differentiates shared and non-shared memory locations and associates each shared location with a set of locks. It may be possible to process such a model to generate directives for private and guarded regions, an approach suggested by other researchers [29].

5.3.3 The problem of recycling memory

The current runtime for checked execution does not recycle memory when it is freed. Freeing memory represents a challenge for the approach because the call to free() happens on a single filament but effectively revokes access to the memory for all filaments in the system. Since filaments communicate asynchronously, the runtime has no facility to ensure that all of the filaments drop permissions to the memory, introducing the possibility that undetected data races will occur in the future.

Adding dependency tracking and finalization to the runtime will greatly reduce this problem for request-oriented systems. When requests are finalized the runtime is able to prove that no filament within the system has maintained a reference to the memory. Therefore, freeing memory during finalization should be safe assuming that no references were maintained within the modules behind a tollgate.

In future work, we believe that we can address the problem of freeing memory by using the same permission checks for memory allocations and frees that we use for writes. One consequence of this approach is that it will be difficult to free data in immutable regions without additional static inference to ensure that the free is invisible to the immutable policy.

Frees in atomic regions are also problematic. However, we make no claim to detect data races on blocks in atomic regions, since these regions are unsound by definition. Therefore, we can support frees in these regions by ensuring that the memory freed in atomic regions is used to create new atomic blocks.

Recycling within a region is safe for regions protected by the other policies outlined in this paper. However, recycling within a region does not allow us to handle the destruction of entire regions. This question must be addressed on a region-by-region basis. The private region for a filament is destroyed when the filament terminates. Once the filament has terminated it is possible to give its memory to any other filament. This is simply a case of serial thread confinement where the hand-off event is tied to the termination of the first owner.

A similar argument holds for guarded regions. The destruction of the region represents a write. Therefore, the lock must be held at the time of the destruction. Transfer of the memory is permissible after the corresponding unlock call as long as the runtime ensures that the required memory barriers are in place.

Recycling memory that was part of a transferable region does not introduce the risk of undetected data races. However, there is a risk that a filament could use a dangling pointer to obtain access to a region. In this case, there is a risk that the runtime would misidentify the use of the dangling pointer as a transfer conflict. We could provide a more specific error by changing the directives, requiring threads to provide a region identifier when requesting a transfer. This would increase the annotation effort for most applications.

5.3.4 Reducing the cost of mprotect()

The benchmarks indicate that changing the permissions of pages accounts for roughly half of the overhead added by the dynamic checker. It is likely that most of this time is spent within the mprotect() system call. Most of the calls to mprotect() change the permissions of only a single page. However, permissions changes for multiple regions tend to be clustered at specific points in the code. The large number of calls to mprotect() is caused by the fragmentation of regions and the inability of the current directives to express that the permissions for multiple regions should change simultaneously.

There are at least three opportunities for optimization within the Linux kernel. First, it would be helpful to have a version of the mprotect() system call that could change the permissions of a set of non-contiguous pages. Second, it would be helpful to optimize the kernel to better cope with frequent permissions changes. Current Linux kernels maintain two copies of the permissions for a page, one in the page table entry and one in the page's vm_area_struct structure. All of the pages in a vm_area_struct must have the same permissions. Therefore, many of the mprotect() calls cause vm_area_structs to be split and/or merged.

The testing indicates that the overhead for locating, splitting, and merging vm_area_structs accounts for 30%-70% of the execution time of typical mprotect() calls. This overhead could be reduced by allowing pages in a vm_area_struct to have different permissions. Finally, the x86 version of the kernel currently flushes the entire TLB for each mprotect() call. Benchmarks indicate that flushing individual TLB entries for the affected pages would increase the time spent in mprotect(), but would in theory reduce the TLB miss rate after the call completed.

5.3.5 Policies to avoid transfer conflicts

The runtime generates an error called a transfer conflict when two filaments attempt to obtain conflicting access to the same transferable region. Additional checking, often at the hardware level, ensures that filaments do not access blocks in transferable regions without requesting access to the region. When these checks are combined we can be certain that there are no undetected data races for blocks within transferable regions.

If the directives provided by a developer are unsound, it is possible for transfer conflicts to be detected on some runs of the application and not others. This happens because we do not have directives and policies that relate get and release operations of transferable regions. These policies are difficult to generalize; they always depend on a specific happens-before relationships in the program, they must ensure that no two filaments rely on the same happens-before relationship, and they often depend on knowledge of the application's state, such as the number of scanlines completed in frames in x264. When we have examined more concurrent systems it may be possible to generalize these policies.

It would be possible to detect transfer conflicts that occur when there is no happens-before relationship between the releasing and getting threads. However, these checks cannot avoid false negatives, since it is possible that the releasing and getting threads will establish a happens-before relationship for some other purpose.

5.3.6 Using directives to improve efficiency

We believe that it is possible to use the knowledge provided by the directives to improve the performance of applications running on non-traditional memory models. For example, the dynamic checker could allocate blocks in private regions from fast, local memory on a NUMA system.

The directives may also be useful when implementing software transactional memory [91]. During normal operation the transactional memory system would rely on the directives to maintain the read and write set for transactions. It would manipulate the page tables of filaments to detect accesses that were not predicted by the directives. The system would respond to faults by adding the block to the current transaction's read and write set, and could also log these faults to allow programmers to improve the performance of their code by increasing the number and accuracy of the directives.

5.4 Conclusion

This chapter discussed the design of two runtimes that use information provided by the directives described in Chapter 3. The first runtime uses the directives to provide operations that can stop and undo the effects of requests. These operations can be invoked while requests are active, allowing developers to implement behaviors that improve the attentiveness of their system. Developers can use these operations even when the system incorporates third party modules by placing directives at the module boundary. However, the operations will compromise the system's consistency if the information provided by directives is inaccurate. Therefore, Section 5.2 discusses the design and implementation of a second runtime that checks the accuracy

of directives as the system executes. Together these runtimes allow developers to balance the efficiency of their system, its level of attentiveness, the implementation effort of adding directives, and the reliability of redirection.

Chapter 6

Conclusion

Chapter 2 defined attentiveness, a quality attribute that describes the behavior of systems as they redirect work in response to changes in the priorities of their clients. The concept of attentiveness is not entirely novel: attentiveness is similar to the quality attribute addressed by Petra-flow [33], a framework for developing systems that remain responsive while relying on variable resources, such as network bandwidth. Like the work described here, Petra-flow can redirect work in progress in response to external events. However, Petra-flow requires developers to implement their systems using specialized state variables and to express the relationships between these state variables and tasks. Petra-flow's approach can be highly efficient, but it requires detailed analysis and extensive rework of third-party code incorporated in the system.

The approach outlined in this document differs, trading off the efficiency of redirection to facilitate the incorporation of third-party code. The approach must accommodate the patterns of state management that are already present in the code. However, these patterns often do not express all of the intent needed to ensure the soundness of the system during redirection. Therefore, the runtime requires developers to insert the directives described in Chapter 3 to provide some of the missing intent. In most cases the code can be incorporated without change, and analysis can be confined to the code's interface. The runtime system, described in Chapter 5, propagates the information provided in the directives along the control flow of threads in the system. The runtime is able to provide low-level operations that allow attentiveness concerns to be isolated, as illustrated in the design outlined in Chapter 4. The approach allows developers to improve the attentiveness and efficiency of systems incrementally by adding additional directives to provide additional information to the runtime system.

Developers could use a similar approach of documenting intent in directives, using a runtime to gather information from the directives as the system executes, and using the runtime to modify the system's execution to achieve other goals:

- Refactoring single-threaded code to use multiple threads. The runtime support described in Chapter 5 can be applied without modification to assist developers when adding threads to existing single-threaded systems. When refactoring these systems developers attempt to identify long computations, choosing computations that can proceed with minimal synchronization with the rest of the system. However, developers may miss some shared state, introducing data races. The runtime support can identify shared state quickly, reducing the time to find defects in the refactoring.
- Crash protection. It may be possible to recover the state of a system after it fails while processing a request by employing the redirection approach described in Chapter 5. The approach assumes that the request caused the failure by corrupting one or more of the regions. These regions would be restored when the request was redirected, reviving the system. If requests are logged the system may be able to recover when a prior request is responsible for the failure by replaying the other requests in the log from a known good checkpoint.

- Unit testing of toolkit-based applications. Testing of toolkit-based applications is often hampered by the low-level nature of the events used to submit requests to the system. Tools are often unable to reliably submit requests due to subtle timing dependencies and changes in the system's user interface. Tools can avoid this problem in applications that employ the design discussed in Chapter 4 to inject requests directly at the scheduler.
- Limiting access to sensitive in-process data. Many systems must limit access to data depending on the state of their clients. For example, an IMAP server must not provide email messages to a client before it has authenticated. Defects in the authentication system may allow unauthenticated clients to gain control of the IMAP server. Once these clients have gained control of the system they may be able to access the messages directly. The data access policies developed in Chapter 3 to detect data races could be extended to consider the level of authentication when threads attempt to access regions. With additional kernel support to prevent clients from bypassing the runtime, it would be possible to create a runtime system that would enforce these policies. The runtime system would stop a compromised system before an unauthorized client gained access to data.
- Monitoring distributed tasks. Developers frequently find it difficult to debug tasks that are submitted to large clusters of computers. In these clusters failures occur on nodes that are not under the direct control of the system's developers. In addition, it is frequently difficult to differentiate three types of failures: failures caused by unreliable nodes, intermittent failures caused by software defects, and reproducible errors caused by software defects. The process of differentiating these failures is complicated by the cluster management system, which restarts jobs automatically in an attempt to cope with defective nodes. As a result, the cluster management system may mask intermittent failures and may delay reporting reproducible failures. In addition, nodes occasionally fail slowly or silently, leading to further delays. The combination of directives, mediators, and runtime support developed in this research could address these problems by providing additional information to the cluster management system and the developers. When software running on a node fails due to the violation of a directive, the cluster management system could forward the report to developers, allowing them to decide if the failure was due to a hardware fault or a software defect. In addition, the cluster management system could compare these reports, allowing it to avoid repeatedly restarting a system that exhibits a reproducible failure. As a result, reproducible failures will be reported with less delay, saving the time of both developers and the cluster. Finally, the runtime provides an opportunity to monitor the progress of nodes, potentially allowing the cluster management system to detect nodes that are not making acceptable progress and restarting them to avoid blocking the overall task.

The goals described above share three features. First, achieving the goals depends on intent that is not expressed in the system's implementation. Directives provide developers a way to document this intent. Second, the goals are difficult to assess by a static analysis of the system, but can be mechanically checked during the system's execution with modest impact on performance. Finally, the goals are important, making it plausible that developers would be willing to accept a modest loss in performance to make incremental progress toward the goal. Therefore, adopting the approach outlined in this research may allow developers to make incremental progress toward the goal, trading off the efficiency of the system for incremental effort in placing directives.

Bibliography

[1] Marwan Abi-Antoun, Nariman Ammar, and Thomas LaToza. Questions about object structure during coding activities. Technical Report CMU-ISR-10-102, School of Computer, Science Carnegie Mellon University, January 2010.

[2] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[3] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: checking data sharing strategies for multithreaded C. *SIGPLAN Not.*, 43(6):149–158, 2008.

[4] Apple. Concurrency programming guide. *On Web*, August 2009.

[5] Apple Computer, Inc. *Inside Macintosh: Macintosh Toolbox Essentials.* Addison-Wesley, October 1992.

[6] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[7] Ronald M. Baecker and William A. S. Buxton. The Star, the Lisa, and the Macintosh. In Ronald M. Baecker and William A. S. Buxton, editors, *Readings in human-computer interaction: a multidisciplinary approach*, pages 649–652. Morgan Kaufmann, 1987.

[8] T.P. Baker and A. Shaw. The cyclic executive model and Ada. In *Real-Time Systems Symposium*, pages 120 –129, December 1988.

[9] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology. Technical Report MSR-TR-2004-08, Microsoft, Inc., January 2004.

[10] Raymond E. Barber and Jr. Henry C. Lucas. System response time, operator productivity, and job satisfaction. *Commun. ACM*, 26(11):972–986, 1983.

[11] L. Bass and B. E. John. Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software*, 66(3):187–197, June 2003.

[12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95*, pages 1–10. ACM Press, 1995.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08*, pages 72–81. ACM, 2008.

[14] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, 2008.

[15] M. Butler, C. Ferreira, and M. Ng. Precise modelling of compensating business transactions and its application to BPEL. *Journal of Universal Computer Science*, 11(5):712–743, 2005.

[16] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *CHI '91*, pages 181–186. ACM, 1991.

[17] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The ATOMOΣ transactional programming language. In *PLDI '06*,

pages 1–13. ACM Press, 2006.

[18] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: limited specifications for analysis and manipulation. In *ICSE '98*, pages 167–176. IEEE Computer Society, 1998.

[19] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. *ECOOP 2003 – Object-Oriented Programming*, pages 59–67, 2003.

[20] Java Community. JSR-133: Java™memory model and thread specification. Technical Report 133, Java Community, August 2004. August 24, 2004.

[21] Alan Cooper and Robert M. Reimann. *About Face 2.0: The Essentials of Interaction Design*. Wiley, March 2003.

[22] Gordon V. Cormack. A calculus for concurrent update. Technical Report CS-95-06, University of Waterloo, 1995.

[23] M. Crispin. Internet Message Access Protocol - version 4rev1, 2003.

[24] Cyrus, 2010. http://www.cyrusimap.org/.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04*, pages 10–10. USENIX Association, 2004.

[26] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *20th international symposium on distributed computing*, pages 194–208, 2006.

[27] Dovecot, 2009. http://www.dovecot.org.

[28] Jakob Engblom, Andreas Ermedahl, Mikael Sodin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 2001.

[29] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE workshop on dynamic analysis*, pages 24–27, May 2003.

[30] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[31] Marc Evers. A case study on adaptability problems of the separation of user interface and application semantics. Technical Report 99-14, University of Twente, Dept. of Computer Science, Software Engineering Group, 1999.

[32] Donald G. Firesmith. Common concepts underlying safety, security, and survivability engineering. Technical Report CMU/SEI-2003-TN-033, Software Engineering Institute, Carnegie Mellon University, 2003.

[33] George Forman. *Obtaining Responsiveness in Resource-Variable Environments*. PhD thesis, Dept. of Computer Science & Engineering, University of Washington, 1996.

[34] Amy Fowler. A Swing architecture overview. http://java.sun.com/products/jfc/tsc/articles/architecture/, 2001.

[35] Keir Fraser and Fay Chang. Operating system I/O speculation: How two invocations are faster than one. In *USENIX 2003 Annual Technical Conference*, pages 325–338, June 2003.

[36] Hania Gajewska, Mark S. Manasse, and Joel McCormack. Why X is not our ideal window system. *Softw. Pract. Exper.*, 20(S2):137–171, 1990.

[37] Emden R. Gansner and John H. Reppy. *A multi-threaded higher-order user interface toolkit*, pages 61–80. John Wiley & Sons, Inc., 1993.

[38] Hector Garcia-Molina and Kenneth Salem. SAGAS. In *SIGMOD '87*, pages 249–259. ACM Press, 1987.

[39] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: why reuse is so hard. *Software,*

IEEE, 12(6):17 –26, nov 1995.

[40] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, May 2006.

[41] Arthur P. Goldberg. Transparent recovery of Mach applications. In *USENIX Mach symposium*, pages 169–184, 1990.

[42] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *7th international conference on Very Large Data Bases*, pages 144–154. IEEE Computer Society, 1981.

[43] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP working conference on modelling in data base management systems*, pages 365–394, 1976.

[44] Aaron Greenhouse. A programmer-oriented approach to safe concurrency. Technical Report CMU-CS-03-135, School of Computer Science, Carnegie Mellon University, May 2003.

[45] Aaron Greenhouse and John Boyland. An object-oriented effects system. *ECOOP 99*, pages 668–668, 1999.

[46] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *ICSE '02*, pages 453–463. ACM, 2002.

[47] GTK+, 2009. http://www.gtk.org.

[48] Jan L. Guynes. Impact of system response time on state anxiety. *Commun. ACM*, 31(3):342–347, 1988.

[49] Scott C. Hale. FlashLight: A dynamic detector of shared state, race conditions, and locking models in concurrent Java programs. Master's thesis, Air Force Institute of Technology, March 2006.

[50] T. J. Halloran. Analysis-based verification: A programmer-oriented approach to the assurance of mechanical program properties. Technical Report CMU-ISR-10-112, Carnegie Mellon University, Institute for Software Research, May 2010.

[51] Robert Halstead. New ideas in parallel Lisp: Language design, implementation, and programming tools. *Parallel Lisp: Languages and Systems*, pages 1–57, 1990.

[52] Graham Hamilton. Multithreaded toolkits: A failed dream? *Graham Hamilton's Blog*, October 2004. http://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html.

[53] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., 1985.

[54] Tim Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[55] Stephen J. Hartley. "Alfonse, wait here for my signal!". *SIGCSE Bull.*, 31(1):58–62, 1999.

[56] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93*, pages 289–300. ACM Press, 1993.

[57] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[58] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[59] Sergey Ignatchenko. STL implementations and thread safety. *C++ Report*, 10(7):61–64, July 1998.

[60] Inkscape, 2009. http://www.inkscape.org.

[61] A. Jannesari, Kaibin Bao, V. Pankratius, and W.F. Tichy. Helgrind+: An efficient dynamic race detector. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–13, 2009.

[62] Ralph E. Johnson. Components, frameworks, patterns. In *SSR '97*, pages 10–17. ACM Press, 1997.

[63] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[64] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[65] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[66] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[67] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[68] Thomas D. LaToza. Answering control flow questions about code. In *OOPSLA companion '08: companion to the 23rd ACM SIGPLAN conference on object-oriented programming systems languages and applications*, pages 921–922. ACM, 2008.

[69] R. Greg Lavender and Douglas C. Schmidt. Active Object. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2 (software patterns series)*. Addison-Wesley Professional, June 1996.

[70] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[71] Doug Lea. *Concurrent Programming in Java: Creating Threads.* Addison-Wesley Longman Publishing Co., Inc., 2001.

[72] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Transaction reordering with application to synchronized scans. In *DOLAP '08: proceeding of the ACM 11th international workshop on data warehousing and OLAP*, pages 17–24. ACM, 2008.

[73] Donald Michie. "Memo" functions and machine learning. *Nature*, 218(5136):19–22, April 1968.

[74] Brad A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *UIST '91*, pages 211–220. ACM Press, 1991.

[75] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07*, pages 68–78. ACM Press, 2007.

[76] The Open Group. *The Single UNIX Specification*, 1997.

[77] Neil O'Toole. Interruptible RMI. *On Web.*, February 2010. https://interruptiblermi.dev.java.net/.

[78] Christopher R. Palmer and Gordon V. Cormack. Operation transforms for a distributed shared spreadsheet. In *CSCW '98*, pages 69–78. ACM, 1998.

[79] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[80] Amir Pnueli. The temporal logic of programs. In *Foundations of computer science, 1977., 18th annual symposium on*, pages 46–57, 1977.

[81] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP '03*, pages 179–190. ACM, 2003.

[82] Atul Prakash, Hyong Sop Shim, and Jang Ho Lee. Data management issues and trade-offs in CSCW systems. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):213–227, 1999.

[83] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. ISOLATOR: dynamically ensuring isolation in concurrent programs. In *ASPLOS '09: proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192. ACM, 2009.

[84] Michael F. Ringenburg and Dan Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP*

'05, pages 92–104. ACM Press, 2005.

[85] RoundCube 0.4 beta, 2010. http://roundcube.net/.

[86] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97*, pages 27–37. ACM, 1997.

[87] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, 1984.

[88] Nicolás Serrano and Juan Pablo Aroztegi. Ajax frameworks in interactive web apps. *IEEE Software*, 24(5):12–14, 2007.

[89] Julian Seward, Cerion Armour-Brown, Jeremy Fitzhardinge, Tom Hughes, Nicholas Nethercote, Paul Mackerras, Dirk Mueller, Robert Walsh, Josef Weidendorfer, Frederic Gobry, Daniel Berlin, Nick Clifton, Michael Matz, Simon Hausmann, Bryan Meredith, Rich Coe, and Bart Van Assche. *Valgrind Documentation*, release 3.3.1 edition, June 2008.

[90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[91] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95*, pages 204–213. ACM Press, 1995.

[92] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16(3):265–285, 1984.

[93] Ben Shneiderman. *Designing the User Interface.* Addison Wesley, July 1997.

[94] Ben Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Proceedings of the 2nd International Conference on Intelligent User Interfaces*, pages 33–39. ACM Press, 1997.

[95] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (It's time for a complete rewrite). In *VLDB '07*, pages 1150–1160. VLDB Endowment, 2007.

[96] Sun Microsystems, Inc. *Java Remote Method Invocation Specification, Revision 1.10.* On Web., 2004.

[97] Dean F. Sutherland and William L. Scherlis. Composable thread coloring. In *PPoPP '10.* ACM, 2010.

[98] The Glib Team. Glib reference manual. http://developer.gnome.org/doc/API/glib.

[99] Thunderbird, 2009. http://www.mozilla.com/en-US/thunderbird/.

[100] Jin-Min Yang, Da-Fang Zhang, Xue-Dong Yang, and Wen-Wei Li. Reliable user-level rollback recovery implementation for multithreaded processes on Windows. *Software: Practice and Experience*, 37(3):331–346, 2007.

[101] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT '06/FSE-14*, pages 81–91. ACM Press, 2006.

Appendix A

Risk of redirection to intra-component consistency

Developers cannot be expected to recover the consistency of the system's state after redirection. For example, consider the routines shown in Listing A.1 that maintain a doubly linked list. Figure A.1 shows the steps that a system would follow if addAll() were called with two lists, each of which has two nodes. One set of states in the diagram, SS01-SS12, shows an unoptimized execution that follows the order of operations in the code. In this case, its developers could write code to recover the consistency of the system's state after redirection.

However, compilers are free to make optimizations to code during compilation, such as eliminating redundant stores [6]. These optimizations relax the Sequential Consistency model [67], leading to states that could not be reached from a sequential execution of the code. These states are shown in the diagram as OS01-OS10. Most of these states could not be reached from a sequential execution of the source code. As a result, developers have no way of predicting the contents of memory if the addAll() routine were redirected, making it impossible for them to recover the consistency of the system's state. In some cases the head and tail pointers of one of the lists would point to the wrong nodes. In other cases the forward and backward pointers in the list would be inconsistent. Therefore, approaches to ensuring intra-component consistency must be pessimistic, preserving consistent states that can be restored in the event of redirection.

There are also problems for inter-component consistency. For example, in the linked list example given above states SS04 and SS09 have intra-component consistency: each of the linked lists is completely consistent. However, one of the nodes has been removed from the source list but has not yet been linked to the destination list. If the system were to stop in one of these states the node would be lost, violating inter-component consistency. In this case the unstated invariant is that in the event of redirection while addAll(L1, L2) is called, every node in L2 either remains in L2 or is added to L1.

```
typedef struct node {
   struct node * prev;
   struct node * next;
} node_t;

typedef struct list {
  node_t * head;
  node_t *tail;
} list_t;

static inline void append(list_t *list, node_t *node) {
   node->next = NULL;
   node->prev = list->tail;
   list->tail = node;
   if (list->head == NULL)
      list->head = node;
   else
      node->prev->next = node;
}

static inline node_t *pop(list_t *list) {
   node_t * const rval = list->head;
   node_t * const pop_next = rval->next;
   list->head = pop_next;
   if (pop_next == NULL)
      list->tail = NULL;
   else
      pop_next->prev = NULL;
   rval->prev = rval->next = NULL;
   return rval;
}

static inline node_t *push(list_t *list) {
    ...
}

void addAll(list_t *restrict dst, list_t *restrict src) {
   if (src != dst)
      while(src->head != null)
         append(dst, pop(src));
}
```

Listing A.1: Code to manage a doubly linked list

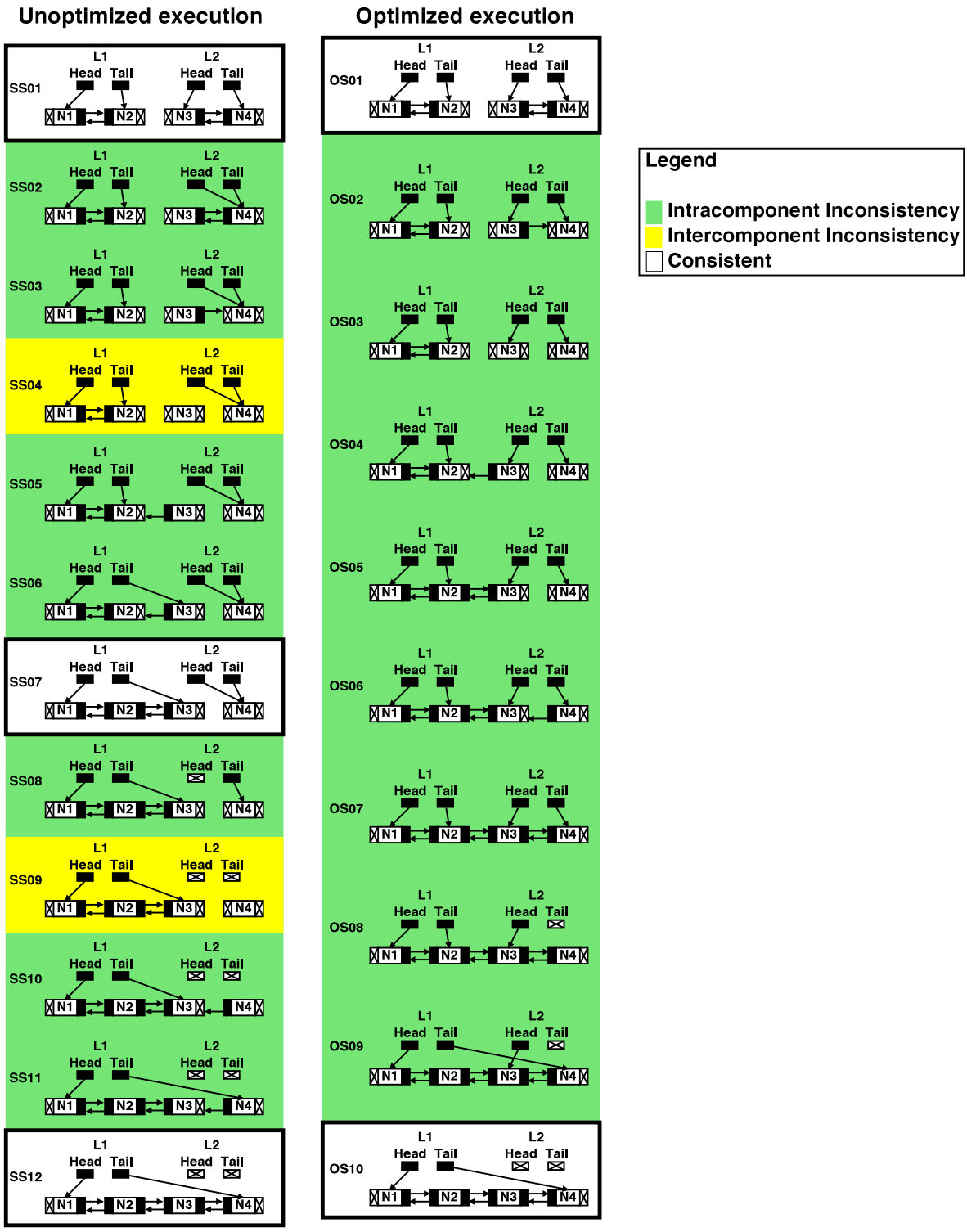


Figure A.1: State changes that occur while moving nodes from L2 to L1. Nodes are moved one at a time. First, the front node of L2 is removed, and then it is added to the end of L1. The process repeats until there are no nodes in L2. When the code is optimized some of the intermediate steps are eliminated, as shown in the right column.