Chapter 4

Programming with Reserves

This chapter describes how to write programs that take advantage of resource reserves to satisfy their timing constraints. It focuses on three main issues: How should reserves be used in an application given that it uses various modules, external servers, and system services in the course of its computation? And also: How should the reservation parameters, particularly the reserved resource usage parameter, be initially chosen? How should they be adjusted given that applications must support different platforms and may have computational requirements that depend on changing input data?

4.1 Overview

This chapter describes the major issues involved in programming with reserves including the design decisions and tradeoffs that a programmer must make. The specific issues addressed are:

- How to structure programs to take advantage of reserves.
- How to map reserves onto a program's structure.
- How reservation parameters should be sized.
- How adaptive programs should adjust reservation levels.

One can think of a program as a graph of computational nodes, and each computational node has a reserve associated with it. Determining exactly what reserves should be allocated, what their reservation parameters should be, and how reserves should be associated with these computations involves design decisions that impact the program structure.

For example, the programmer must decide whether applications that depend on each other will explicitly negotiate timing requirements among themselves for the specific services they provide to each other. The alternative is to allocate resource reserves for their combined activity and then pass those reserves along as the abstract "activity" passes from one to the other. In the first case, the partitioning of requirements and the explicit specification of timing requirements for each computation in each application creates a great deal of bookkeeping that has to be done. In the latter case, the requirements summarize the entire activity without specifying each detail along the way. As long as each phase of the activity adheres to a few rules such as not introducing unnecessary delays into the overall activity, the same high-level timing requirements can be satisfied without excessive dissection of the programs.

Another design decision addressed here relates to the flexibility of applications that use reserves. Hard real-time applications would typically specify fixed reservation parameters. Adaptive programs might be able to monitor their resource usage and adjust reservation parameters to fit their behavior over time. They might even be able to select different algorithms with different semantics and different performance characteristics to tune their computation time.

Finally, this chapter addresses the issue of programming with multiple resources. This requires applications to be broken into sub-computations at points where different resources are required. Coordinating resource reservations on multiple resources to satisfy end-to-end timing constraints requires careful design. Two approaches using reserves are described.

4.2 Using reserves in application design

This section focuses on the structure of applications and how reserves fit into that structure. Programs are considered to comprise one or more concurrent activities. Each activity might have a thread associated with it, and each activity has a call graph describing the subroutines that are called by each subroutine. The call graph is extended to include calls to external servers or system services made by each subroutine.

The following sections describe these extended call graphs and address the coordination of reserves between reserved modules, reserved clients and servers, and reserved operating system services.

4.2.1 Program structure

To understand the timing constraints and resource requirements, one must consider the structure of application code. This section describes how an application might be divided into separate activities. It describes how a periodically executed computation in an activity might be broken down into sub-computations by splitting computations at procedure calls, remote procedure calls (RPCs), and system traps. The result is like a call graph that includes "calls" to servers and to the operating system.

For the purpose of this analysis, an activity is defined to be an abstract thread of control that starts out in a process and moves in and out of user-level servers and the operating system as calls are made to those servers and the system. This is similar to threads traversing objects in Clouds [26]. Such an abstract thread model corresponds to a synchronous programming style, which is in contrast to an asynchronous programming style where activities are essentially “forked” by making asynchronous service requests to external servers or kernel primitives. For example, consider an animation application that synthetically generates animation frames in real time. The application consists of two activities: one to generate and display animation frames and one to process user interface events such as requests to resize the animation window.

Each of these activities may call modules in the same address space, make RPCs to servers, or make system calls. By this definition, when a (synchronous) RPC is made to a server, the activity “moves” to the server for the duration of the server’s computation and then the activity returns to the client when the call returns. If the server were to call another server synchronously, the activity would move to the second server for the duration of the call. The same is true of a system call. When a system call is made, the activity “moves” to the operating system and returns when the call returns.

An activity may be periodic. For example, consider the frame generation activity of the animation application. Suppose this activity originates in a subroutine (called process_frame) that is invoked periodically every 33 ms to process and display frames. Now suppose process_frame calls generate_frame and display_frame, which eventually performs an RPC to a window system server that accesses the frame buffer. Figure 4-1 shows an example call graph rooted at the function process_frame.

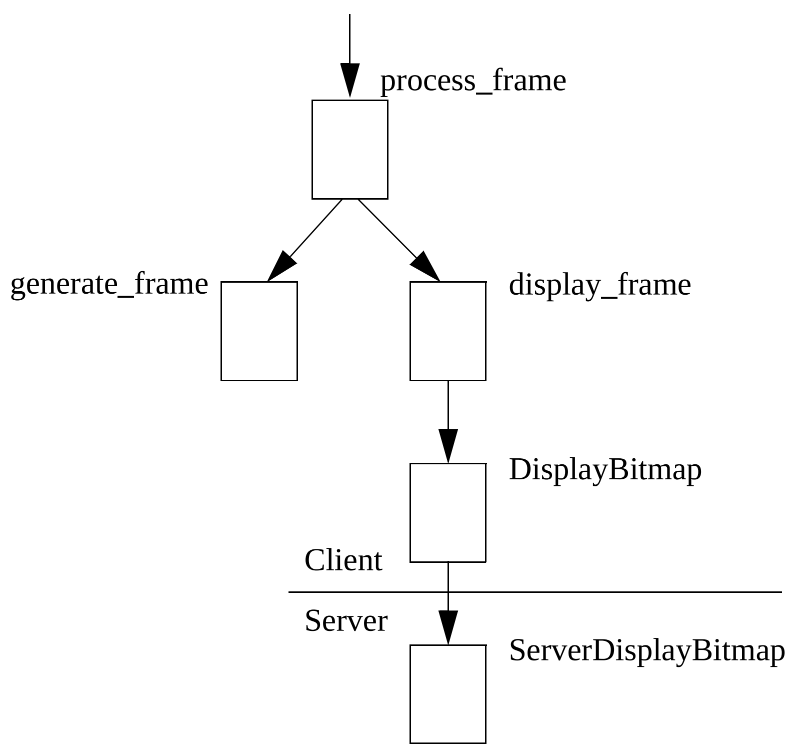

Figure 4-1: Call Graph for Frame Generation and Display

This call graph includes an RPC from the client to the server. The DisplayBitmap subroutine makes the RPC in the client, and the ServerDisplayBitmap subroutine in the server continues the activity. Thus, this graph captures all of the sub-computations of the animation activity.

4.2.2 Reservations for periodic computations

Given that the process_frame subroutine shown in Figure 4-1 is invoked periodically, the thread would “release” the computation periodically by using a while loop with a delay primitive or by setting a period attribute in the case of RT-Mach’s periodic threads [125]. Thus, the activity has an initial release time and a period parameter. To associate a processor reserve with this activity requires that a reserve be allocated with a start time and period that corresponds to that of the process_frame activity. It is possible to bind a periodic thread that attempts to execute its computation every 40 ms to a reserve that has a reservation period of 50 ms. This is not recommended, however, because the resources would not necessarily be available when the activity was released.

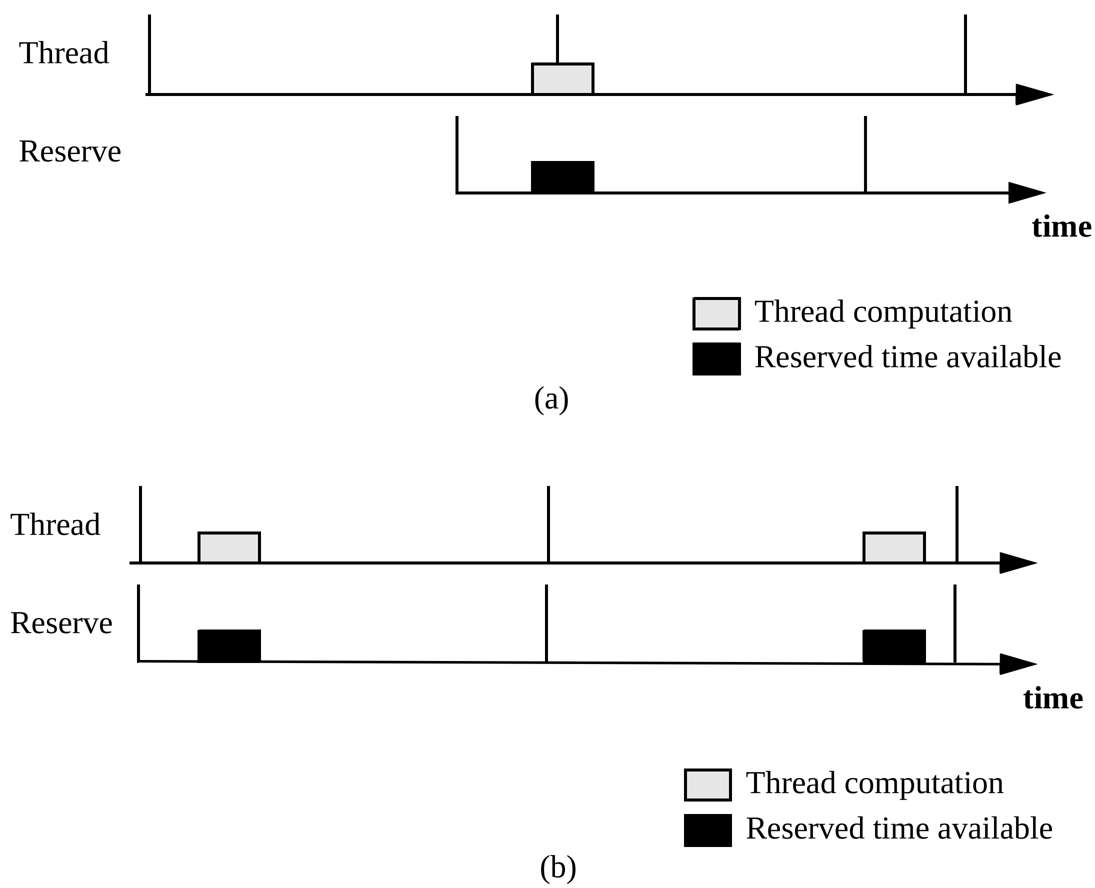

Figure 4-2: Thread and Reserve Out-of-Phase

Figure 4-2(a) illustrates a case where the thread period is not aligned with the reserve period, resulting in undesirable delays. In part (a) of the figure, the reserved computation time is not available until near the end of the thread's period. Thus the thread cannot start running until the very end of its period, and it misses its deadline at the end of the period. The problem is that the availability of the reserved time did not match the availability of the thread. Figure 4-2(b) shows the case where the thread period and reservation period are synchronized. This means that the thread will be ready when the reserved computation time is available, and the reserve guarantees that the reserved computation time will be available by the end of that period, so the thread is assured of being able to complete.

4.2.3 Localized reserve allocation

Consider the resources required in each node of the call graph in Figure 4-1. Assume that generate_frame requires only the processor. For nodes under display_frame, assume the frame buffer is mapped into the window system server's address space and that the processor is the only resource required.

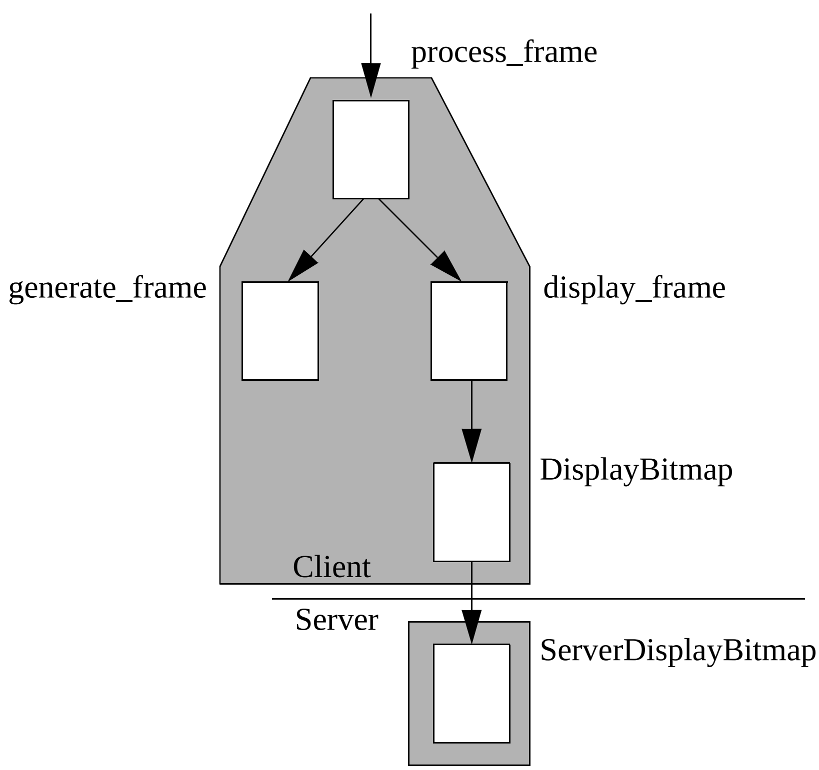

Figure 4-3: Call Graph with Separate Client and Server Reserves

With these assumptions, one approach to allocating reserves for the sub-computations would be to allocate a processor reserve for all of the nodes in the animation application and another processor reserve for the nodes in the window system server. Thus, the server would have a reserve allocated for each of the clients holding open connections to it. This approach is necessary in the case where the server resides on a remote machine, but it may be preferred even when both utilize the same processor. Figure 4-3 illustrates this approach.

The RPC from the client to the server implies a switch from the client's reserve to the server's reserve. With this approach, the traditional "priority" inheritance mechanism would not be useful because the fact that the server has a reserve allocated internally for the animation client defines the "priority" for the client's request in the server and the client's "priority" does not get propagated. Another mechanism to associate the animation applications RPC request with the appropriate reserve inside the server would be very useful. Such a mechanism might take the "priority" of the server-allocated reserve to be associated with the animation client and apply it to the thread that will handle the client's request. This is a kind of "priority" inheritance where the server's thread gets the priority of the reserve it allocated for a client instead of getting the priority of the client itself.

Since the server must allocate the reserve for its computation on behalf of a client, it must know what the reservation parameters should be for the reserve. This approach requires the client and server to enter into a dialogue to allow the client to explicitly request a server-specific QOS level, meaning a certain pattern of server operations to be called with certain timing constraints. The server must then map the requested QOS requirements to system resource requirements and decide whether it can acquire the reserves to support that activity. All of this negotiation must be explicit, and that means a client/server interface for negotiating server-specific QOS requirements must exist. Further, the server must have the machinery to map those QOS requirements to system resource requirements.

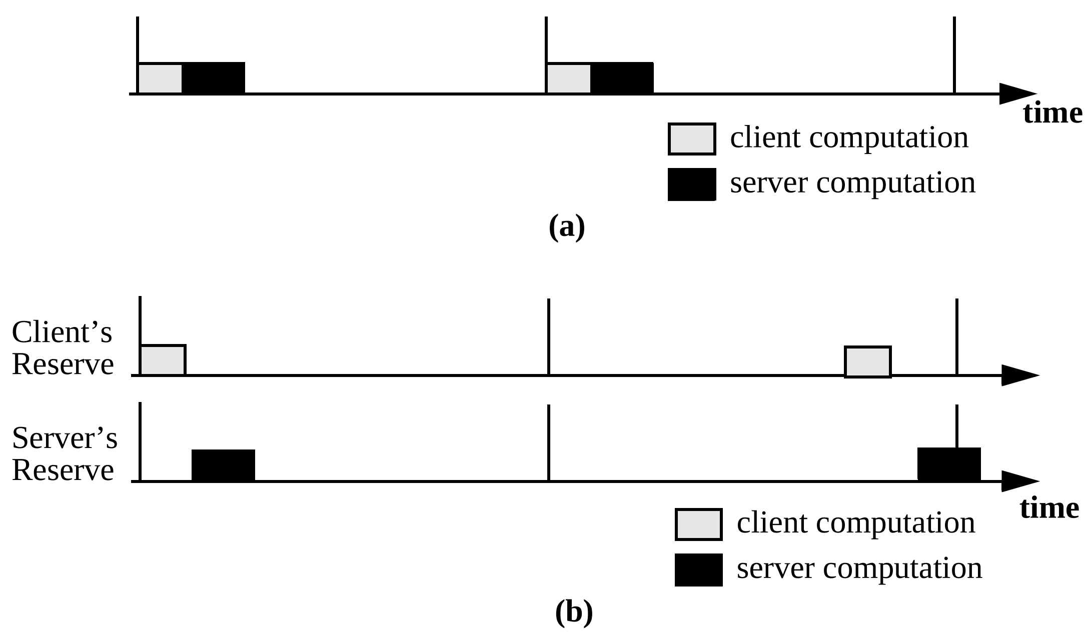

Figure 4-4: Switching Reserve from Client to Server

Another issue is that the timing parameters of the two reserves must be carefully coordinated for the call to be executed smoothly. Essentially, the client's call to the server means that the computation in the server becomes ready, and its reserve must provide the resources for it to execute in a timely fashion after it is ready. The sequence of client computation followed by server computation is illustrated in Figure 4-4(a).

The server's computation time might be available immediately after the call is made, as in the first period of execution history shown in Figure 4-4(b), in which case the deadline for the combined activity is met. But as shown in the second period of the execution history in Figure 4-4(b), the client's computation time may be available very late in its period. The server's computation time may be available earlier in its period but not available so close to the end. It is guaranteed to be available sometime in the period, but not at any particular time. Thus the activity could miss its deadline.

Introducing an intermediate deadline for the client's computation could solve this synchronization problem. Figure 4-5(a) shows the usage pattern of the client and server with an intermediate deadline for the client.

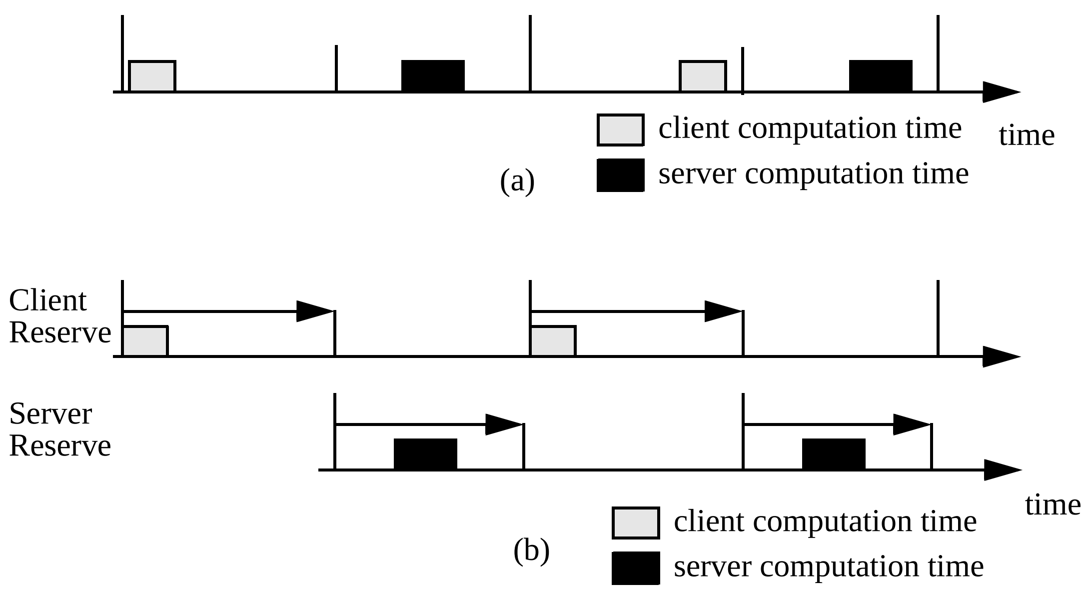

Figure 4-5: Client Requirement with Intermediate Deadline

Using reserves with deadline parameters, Figure 4-5(b) shows how the reserves can be allocated such that the client reserve has its computation time available at the beginning of the client's period with an intermediate deadline halfway through the period. The server's start time is at that intermediate deadline, and it has a deadline that corresponds to the end of the client's overall period. Thus, both activities are guaranteed to synchronize and complete by the overall deadline as desired.

All of the explicit handling of QOS requirements and resource requirements and the careful synchronization of interactions between reserves makes programming clients and servers much more complex. While this may be necessary for designing complex hard real-time systems, for soft real-time systems and less complex hard real-time systems, the approach where resource requirements for clients and servers are folded into one reserve may be better.

4.2.4 Activity-based reserve allocation

Another approach would be to allocate a single processor reserve for all of the nodes in the entire call graph; Figure 4-6 illustrates this approach. Of course, if the server is running on a remote host, this approach may not be feasible since it is not clear how a single processor reserve could be made to represent processor time on two different processors.

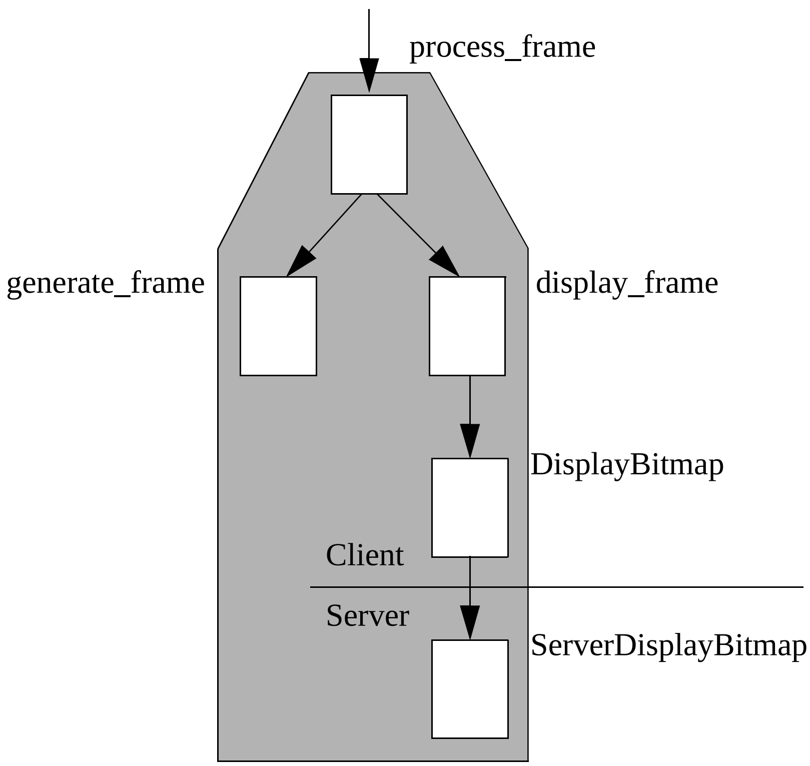

Figure 4-6: Call Graph with One Reserve for All Nodes

Even if the animation application and the server are on the same host running on the same processor, there is a problem that must be addressed in this approach: the server may not be ready to service the RPC call at the time it is issued. In fact, there is a potential “priority” inversion problem associated with such an RPC (where “priority” refers to the ordering of reserved activities by the scheduler rather than an integer priority for a thread). If the

RPC arrives in the server's queue at a time when the server is servicing another client, the server's thread will be bound to the reserve associated with the other client. If that other client is processor-poor, the reserved time may run out during the service, and the server may experience some scheduling delay. If the other client has a reserve that happens to get its processor time very late in its period, there may be a significant delay until the server, running with that other client's reserve, can finish the on-going operation.

To limit the delay the animation application experiences waiting for the server to handle its RPC, a "priority" inheritance protocol [108] must be employed. If the animation client's reserve would be sequenced by the reserve scheduler before the other client's reserve, the server which is using that other client's reserve would be sequenced as if it were using the animation client's reserve. However, for consistency of the usage measurement, it will still charge its usage to the other client's reserve. Then when that service is finished, the server will bind to the reserve of the animation server and that completes the propagation of the animation application's reserve to the server.

For this reserve passing to work best, the RPC should be synchronous, meaning that the client should wait for the result after making the call to the server. With a synchronous RPC, either the client or the server will be charging against the client's reserve whereas with an asynchronous RPC where the client does not wait for the result from the server, both the client and server may be charging against a single reserve at the same time. This is not catastrophic, but it may result in complicated interactions between the client and server.

This approach implies that the client application must request reservation parameters that include the computation time that will be consumed by the nodes residing in the server. This can be done by having the client discover the requirements empirically during runtime, by having the server explicitly provide its resource requirements, or by determining the requirements at design time (this issue is discussed in detail in the next section).

The important point here is that the client and server need not explicitly exchange information about resource requirements if the client allocates the reserve and passes it to the server. In particular, a great deal of complexity can be avoided if the client/server interface does not need to be able to support a complex negotiation of requirements. For legacy systems, this means that existing interfaces need not be radically modified, the only change being the mechanism for passing reserves from client to server.

4.2.5 Coordinating multiple resources

This section describes an issue that arises when an application uses multiple different kinds of resources in different sub-computations. Consider an audio/video player application that reads data stored on a disk and then outputs an audio stream and displays video frames. The player could be structured as three activities: audio playback, video playback, and user interface. The video playback activity would be periodic, reading and displaying a frame every 30 ms.

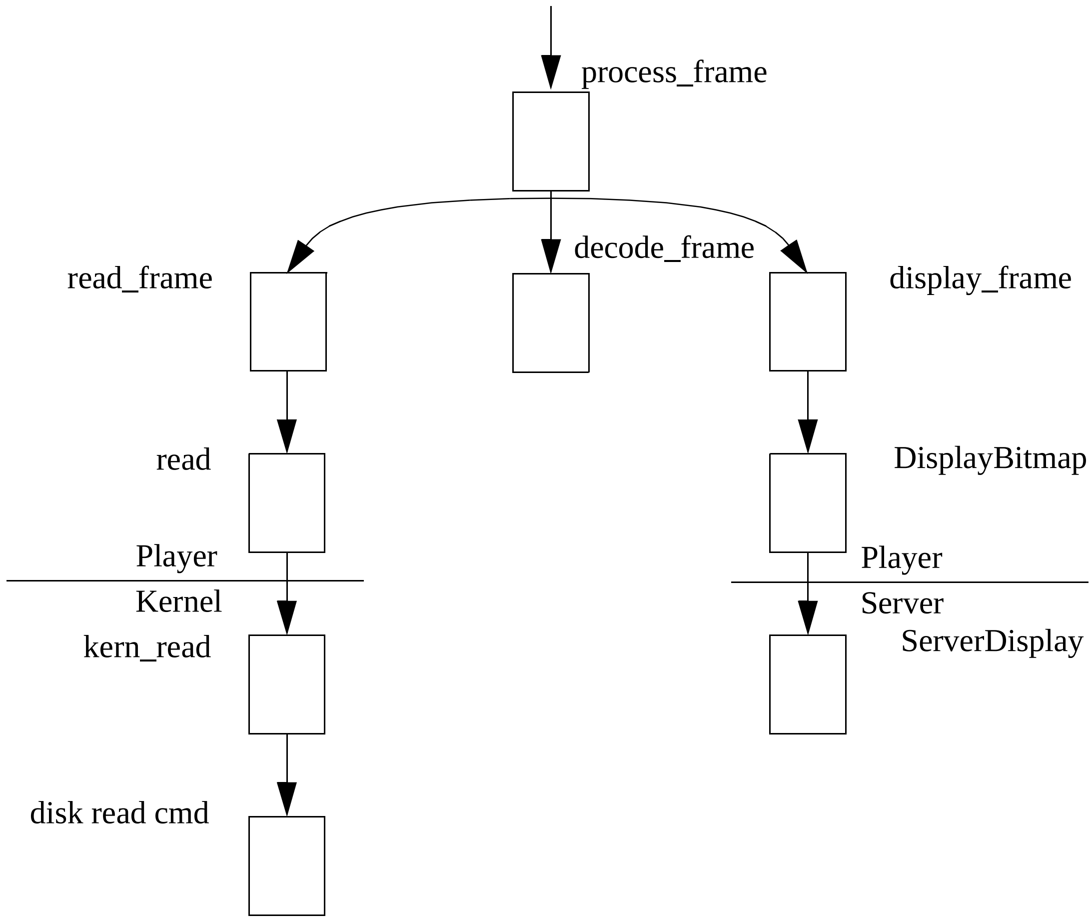

Figure 4-7: Call Graph for Video Playback

Figure 4-7 shows a possible call graph for the video playback activity. The graph is rooted at a subroutine called process_frame, which calls three more subroutines: read_frame, decode_frame, and finally display_frame. In the read_frame routine, calls are made until eventually the program makes a system call and traps into the kernel where more subroutines are called until finally a device command is issued to read data from the disk. This call graph introduces another type of call, referred to as a device command, which is used in addition to the original three types of calls (procedure call, RPC, and system call). The decode_frame routine converts the video frame data to a form that is suitable for display. The last call is to display_frame which is the root for a sequence of calls resulting in an RPC to a window system server which makes additional subroutine calls and finally accesses the frame buffer.

The resource requirements for this call graph include disk access as well as processor time, so a disk reserve is allocated and bound to the disk read node. The other nodes require only the processor, and one approach is to allocate a single processor reserve for all of those. This reserve allocation and binding approach is illustrated in Figure 4-8

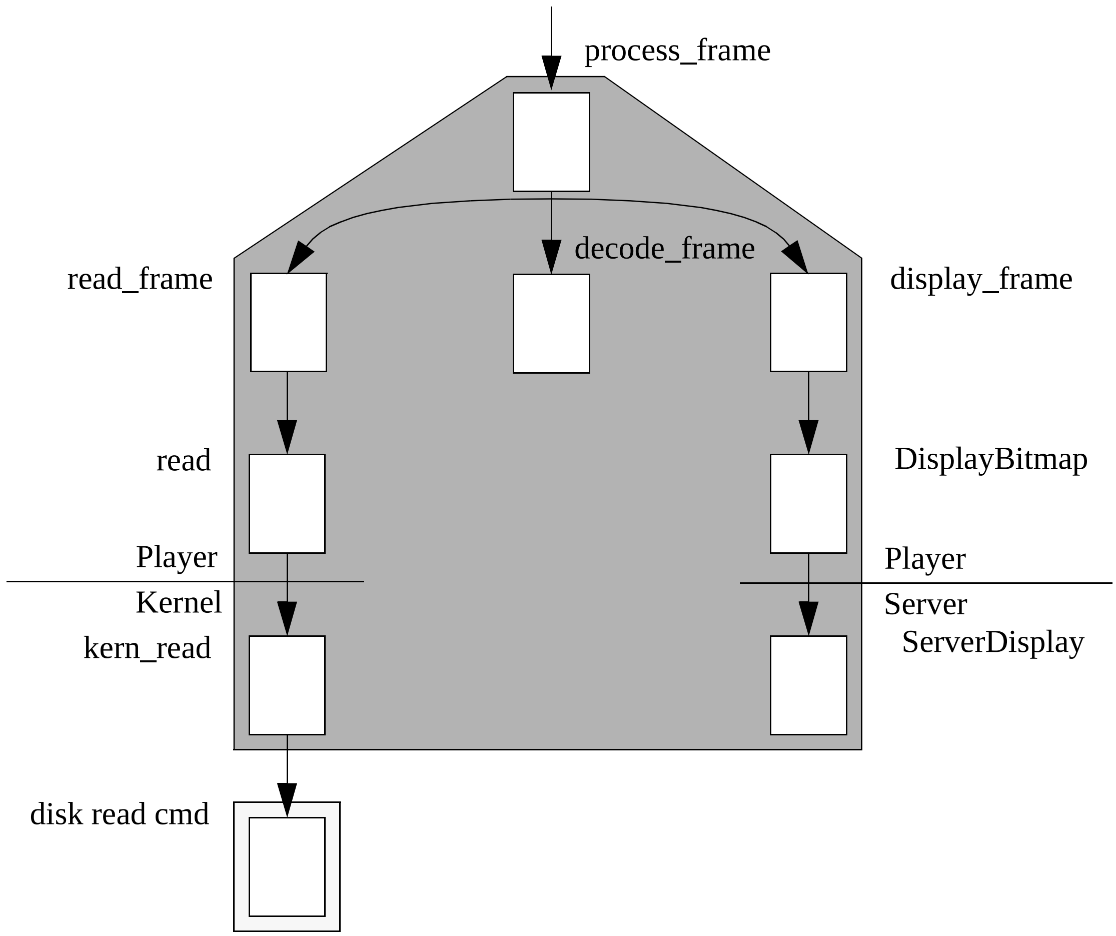

Figure 4-8: Call Graph for Video Playback with Reserves

There is a subtle problem, however, that is related to the synchronization problem between and client and server with localized reserve allocation. The resource usage pattern for the process_frame activity is the following:

- The processor is needed for all the nodes up to where the read command is issued to the disk,
- the disk is required for that read command node,
- all the nodes after that require only the processor.

This resource usage pattern is illustrated in Figure 4-9(a); in every period, the computation first has a processor requirement, then a disk requirement, and then another processor requirement. If a processor reserve and a disk reserve are allocated, the execution pattern may look like the pattern shown in the first period of Figure 4-9(b).

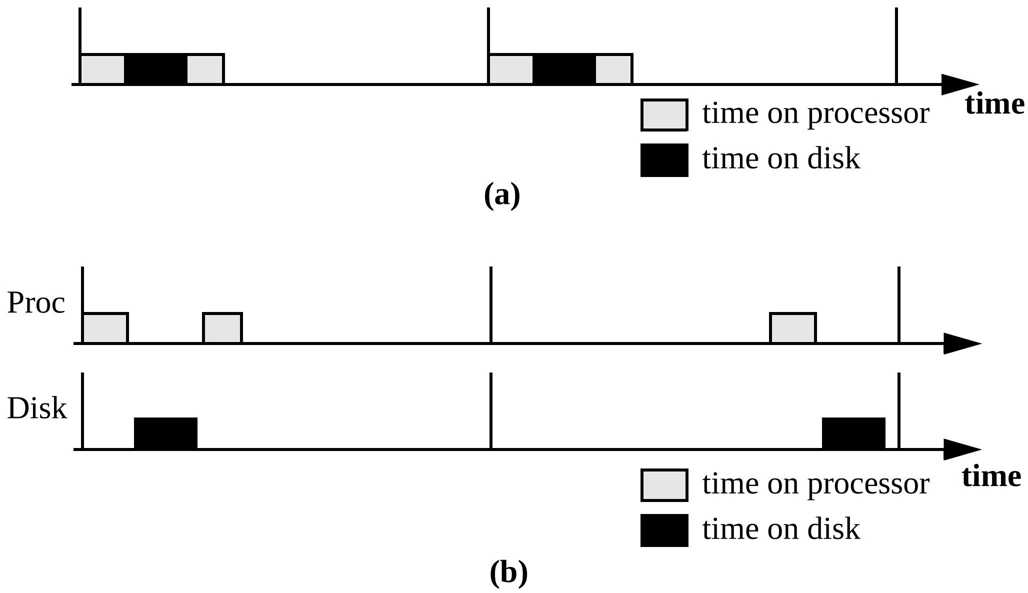

Figure 4-9: Synchronization Problem with Multiple Resources

However, the only guarantee associated with the processor reserve is that the processor time reserved will be available by the end of the period. If that time happens to only be available at the very end of the period, the execution pattern might look like the one in the second 33 ms period of Figure 4-9(b). In that second period, the leading processor requirement is serviced too late, and by the time the disk activity is finished, there is no more time left in the period for the second half of the processor requirement, and the deadline is missed. Worse still, if the reserved disk usage is only available at the beginning of the disk reservation period, the activity will be delayed into the next reservation period and certainly miss a deadline and possibly miss the following deadline as well.

One way to solve this problem is to introduce intermediate deadlines in different stages of the computation to separate sub-computations that use different kinds of resources. For example, Figure 4-10(a) illustrates the usage requirements and new intermediate deadlines for the video playback activity.

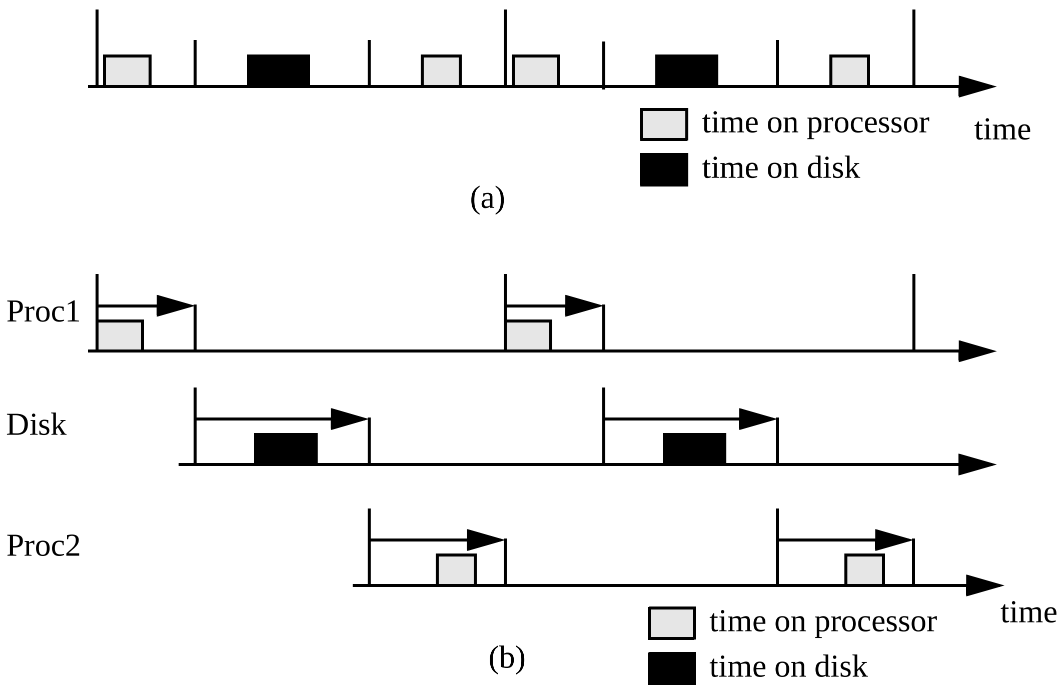

Figure 4-10: Multiple Resources Used with Intermediate Deadlines

A reserve is allocated for each of the three phases of the computation: the leading processor requirements, the disk requirement, and the final processor requirement. Since the end-to-end timing requirement or deadline is divided up into intermediate deadlines for performing the three phases of the overall computation, the reserves that are associated with the phases must have deadline parameters. Figure 4-10(b) shows a timeline for each reservation and how the usage is timed in the three reserves.

So with this approach, two processor reserves (labeled Proc1 and Proc2 in Figure 4-10) and one disk reserve are allocated. The call graph with this reserve allocation and binding appears in Figure 4-11.

This example points out two major factors that influence how reserved computations should be structured and how reserves should be bound to the sub-computations. One factor is the temporal sequence of the resource requirements. Generally speaking, a node in the graph that requires a resource different from its parent acts as a delimiter for grouping computations that can use the same reserve. To minimize the number of reserves required, the application programmer should minimize the number of times computations must switch between required resources.

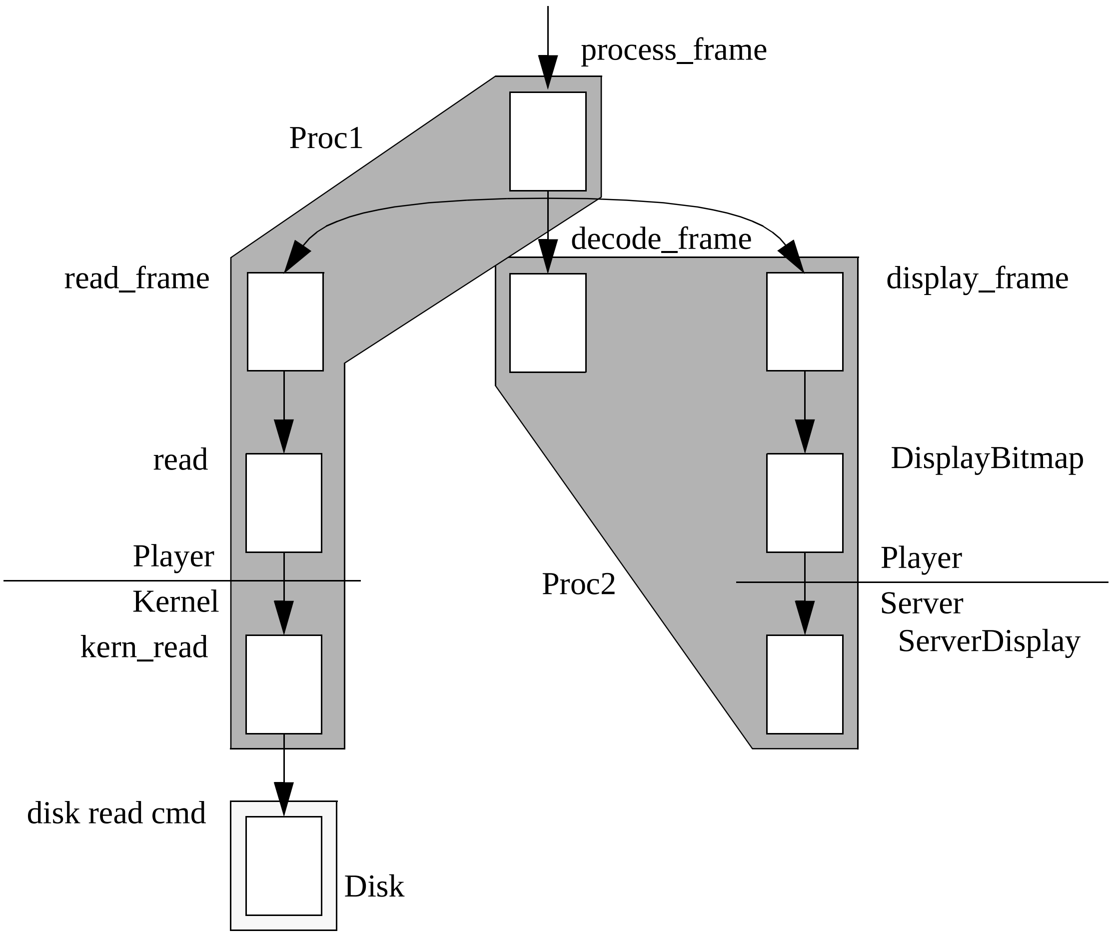

Figure 4-11: Video Playback with Better Reserve/Computation Mapping

The second factor is the spatial organization of the nodes in the system. When a call crosses to another address space or to the operating system, a decision has to be made about whether to switch reserves at that point or not. The system designer has more control over that choice, as described above.

4.3 Sizing Reservations

To use resource reserves, an application must specify appropriate reservation parameters. For hard real-time applications, the reservation parameters would be determined *a priori* by the system designer. For dynamic real-time applications, external agents such as a QOS Manager may suggest or require different reservation parameters during the course of the application's execution. In the dynamic framework, it is important for the application to be aware of the resources required to do the work it needs to do and to be flexible in terms of its timing requirements (how often or under what delay bounds it does the work). In general, two important questions must be addressed:

- How can the resource usage requirements of an application be determined, especially given that the application may be used on different machine types and system configurations?
- How should an application adjust its resource reservations using information about previous performance?

This section deals with the determination of initial resource reservations and adjustment of reservation levels for dynamic periodic activities.

4.3.1 Determining resources required

The first problem to be addressed is how to determine which resources will be needed during the course of the computation and how to determine the initial reservation levels for the various resources required by an activity.

The resources that are needed during the course of a computation will depend on what external services are used by the computation. The list of resources for the computation will be the union of all the resource lists for the transitive closure of external services used by the computation. It is therefore very important for services to be named so that each service can name those services that it uses. And in turn, each service must name those resources that it uses. Then it is possible to find the resources used by the transitive closure of services the application uses.

A potential problem is that different functions offered by a service may use different resources. If a client uses only one function offered by the service, it should reserve only the resources needed for that function rather than the complete list of resources needed by every function the service offers. In this case, it may be useful to consider the resource lists required for "sub-services" or subsets of operations of the service where the subsets are defined to use similar sets of resources.

In any case, the method for ascertaining required resources should be flexible, efficient, and easy to use. Ideally, the system would help to determine the list of resources during an initialization phase of each application. Each time the system encountered an application that required a particular resource but had no corresponding reserve, it would add a reserve of the appropriate type with no reservation parameters to the reserve tree bound to the application. After the initialization phase, the application would have references to the resource that were required by its component computations, even those resources used by servers that were called on its behalf.

Other approaches to determining resources required could be used as well including the following.

- The list of resources could be obtained by sending a query to every server to be used in the computation and having the server provide a list of resources it requires and a list of services it uses. The transitive closure of required resources collected during this query would be accurate as of the server connection time. This method does not require that the system and (possibly more static) documentation remain synchronized with respect to specification of resources used. Upgrades for various system software modules could be made without having to issue new resource list documentation. Also, servers would be free to determine which resources, among many possibilities, would be best to serve that connection given the state of the system and its load at the time the connection was requested. Thus an additional degree of freedom is allowed the servers.

- The list could be found using a database where each server registers the list of resources as well as other services it requires. This makes it possible to write applications that automatically determine the transitive closure of services used and resources required, even if some of those services and resource types did not even exist at the time the application was developed and compiled. One important requirement to make this dynamic method work is that service names and resource types not be hard-coded. Instead, a program should be able to handle and manipulate new service names and resource names with no recompilation.
- The list of resources required by various user-provided services and system services could be static, long-lived, and well documented. The programmer must manually look up all the services and find the transitive closure of services used and then the union of the resource lists of all those services. The major problem with this approach is that the slightest changes to the software for the services may change the list of services used and the resource list, thus making the lists in the manual obsolete and making all the programs written to the specification of the manual obsolete.

4.3.2 Determining initial reservation levels

Once the programmer knows what resources are needed by an activity, she must set up the reserves for those resources and request reservations. Requesting reservations requires that reservation parameters be provided. In the reserve model, a reservation request has parameters for resource time to be reserved and for a reservation period. In many cases, the reservation period will be the same as the period of the activity. This will sometimes be derived directly from user-level quality of service requirements (such as frame rate), and sometimes it will be derived indirectly from user-level requirements. For example, the rate for handling audio packets might depend on the audio sampling frequency, the packet size, and perhaps the system overhead per packet. The resource usage time is more difficult to ascertain. It depends on the platform, the system software, and the data being processed among other things.

One way to get a reasonable estimate as to what the resource usage requirements might be for a given instantiation of an application involves measuring the actual computation that forms the main focus of the application. With one run through a periodic activity, for example, the application could get a fairly good estimate of future computation times using the reservation mechanism's usage measurement features. Another variation on this approach is to use a simple computation to gauge the speed of the machine and/or system architecture, and then use a characterization of the real application's computation expressed in terms of the simple computation to estimate the appropriate reservation level. For example, if the application first ran a SPECint benchmark and knew how much the reserved computation needed in terms of SPECint benchmarks, it could derive the estimate directly.

The following methods could also be used to determine the initial reservation level:

- An application could store in a persistent preferences database some information about reservation levels used in previous instantiations of the application. This information would be a good guess as to what reservation levels should be procured, and it might be possible to maintain a small database to map prior experience with different QOS parameters to reservation parameters. This approach might get much more complicated as more QOS parameters, reservation parameters, and target system architectures are used.
- The initial reservation level could be set to zero or some other relatively small value that is known to be smaller than the actual reservation level, though unknown, that will be required. This approach requires the mechanisms for reservation level adaptation to quickly acquire the feedback on usage that is necessary to set a reasonable reservation level where desired quality of service parameters can be achieved. Initially, the desired QOS parameters will not be achieved and they may never be achieved. These are the major drawbacks of this no-knowledge approach.
- An alternative approach to the zero level initial reservation is to take the maximum reservation level available on the resources at the time the reservation is requested. This has the advantage of having the highest chance of meeting the desired QOS parameters for the application, but the disadvantage is that resource capacity may be unnecessarily tied up and unavailable to other applications requesting reservations. This situation would persist until the adaptation mechanism had the chance to evaluate the situation and make the proper adjustments to the reservation levels.

4.3.3 Measuring performance

An adaptive reserved application should keep track of the resource usage required to perform its computation at each repetition to decide if it has more resource capacity reserved than it needs or if it has too little resource capacity to do its work during each period. It should also keep track of the real-time delay incurred during each repetition of the computation to determine whether the computation was completed within the period or not.

The application can easily measure the real-time delay of a computation by taking a timestamp at the beginning and at the end of the computation. Measuring the resource capacity usage for a computation, however, is more involved, requiring support from the operating system. This support must be more accurate than the traditional logical clock for processor time provided to processes in most operating systems. Logical clocks usually take usage measurements by sampling at clock interrupts to find which process is running and incrementing that process's logical clock as if it had been executing for the entire period. Statistical sampling of this kind, which is inherently inaccurate for short-term measurements, will not provide an application with the clear picture of short-term behavior. Such knowledge of short-term behavior is needed to be able to make suitable adjustments to the reservation parameters.

For the kind of accuracy required for measurements of resource capacity usage, the system must accumulate usage associated with reserves at each context switch. In this context, reserves act as *abstract thread* logical clocks rather than *process* logical clocks. And since the reservation system manages capacity usage for resources other than just the processor, the system must keep usage accumulators for all types of resources, and these must be updated at each context switch on the appropriate resource.

Reserve usage measurements will indicate how an application's actual behavior is related to its reservation. Several possible patterns of behavior are described in the next sections.

4.3.3.1 Balanced applications

An application is balanced with respect to its reservation if the resource usage in each period is fairly constant and the reservation level is at this constant value. (It may be impossible for resource usage to be completely constant for some interesting resources such as processors.)

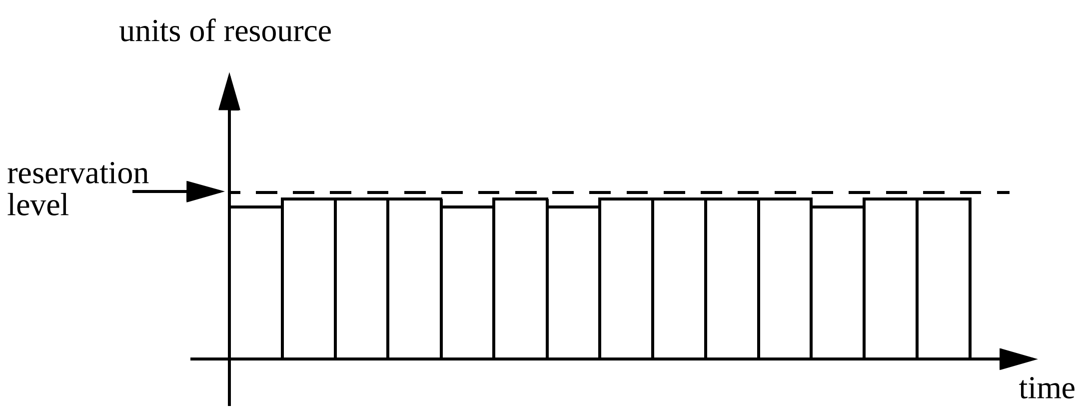

Figure 4-12: Resource Demand Constant and Reserved

Figure 4-12 illustrates a computation's demand on a particular resource over time. Time is on the x-axis, and it is divided into intervals equal to the reservation period. The y-axis is units of resource, e.g. time spent on the processor executing instructions, bytes transmitted, etc. Within each reservation period, the number of units of resource consumed by the computation is measured, and the height of the bar in that interval is the number of units consumed.

In the figure, the number of units of resource required in each reservation period is nearly constant, and the reservation level is slightly more than this constant demand. Therefore, the demand is satisfied by the reservation, and the computation will have the resources to be able to execute completely in each period.

4.3.3.2 Under-reserved applications

An application is under-reserved with regard to a particular resource if its resource usage requirement is greater than its reservation. Two cases are distinguished:

1. worst-case (maximum) resource usage requirement for the computation is greater than the reservation but the average resource usage requirement is less that the reservation, and
2. the average resource usage requirement is greater than the reservation (implying that the worst-case resource usage requirement is also greater than the reservation).

In the first case, the average resource usage requirement is less than the reservation, so over the long term, the application will be able to keep up with its work requirement. The problem is that since the worst-case resource usage requirement is larger than the reservation, the completion of the worst-case computation may be delayed and this may delay or otherwise affect the computations in subsequent periods. If the worst-case computation occurs very infrequently, its negative affects on the overall performance of the application can be minimized or ignored. A human viewer may not even notice an occasional dropped frame during video playback. If the worst-case computation occurs frequently, it may be more difficult to ignore; many dropped video frames would certainly be noticed.

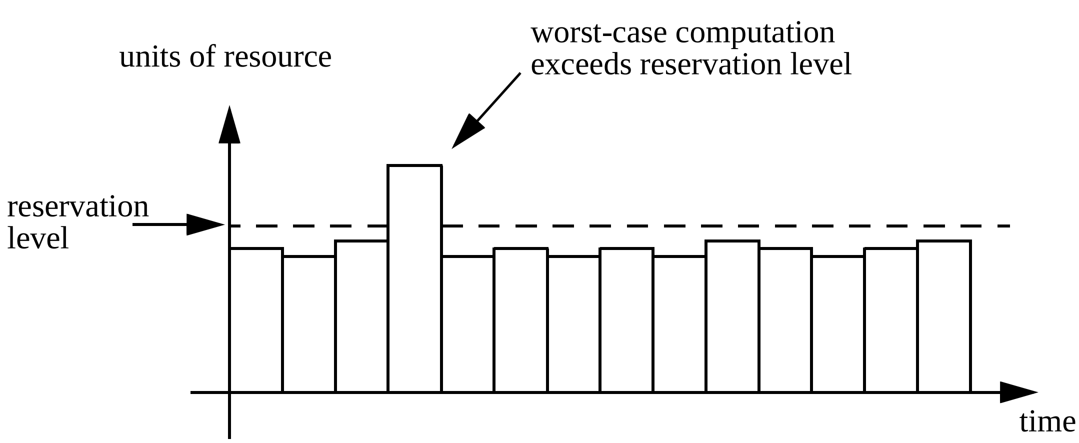

Figure 4-13: Resource Demand Occasionally Exceeds Reservation

Figure 4-13 illustrates this slightly underreserved case. The resource usage requirement in most reservation periods is less than the reservation level for this particular resource. In these periods, the computation will have the resources available to complete. However, there is one period in the illustration (the 4th) in which the resource usage requirement is larger than the reservation. Depending on the system's policy for treating this case, the computation may happen to be completed (using idle time), it may be aborted, or it may extend into the next reservation period, interfering with the completion of the computation which would normally execute in that period.

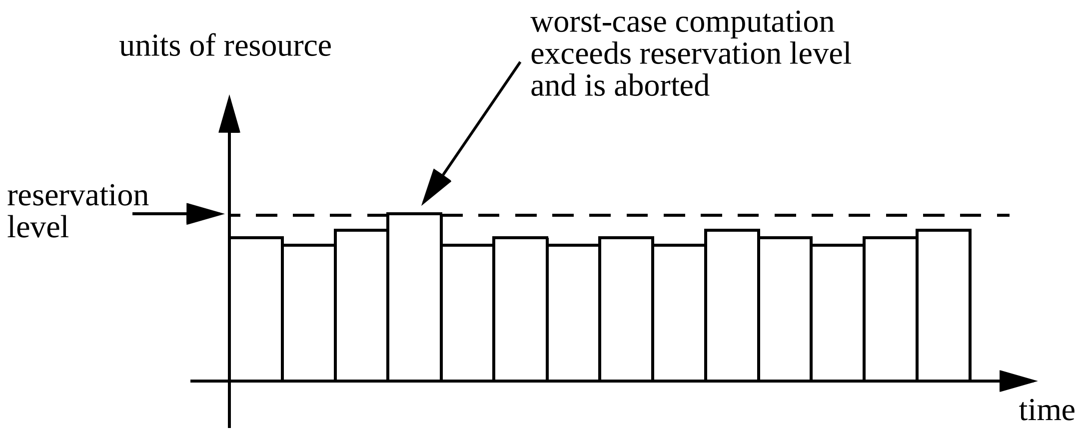

Figure 4-14: Exceedingly Demanding Computation Aborted

Figure 4-14 shows how the usage pattern would look if the computation in the 4th reservation period were aborted. Note that the computations in the subsequent periods are not affected by the aborted computation.

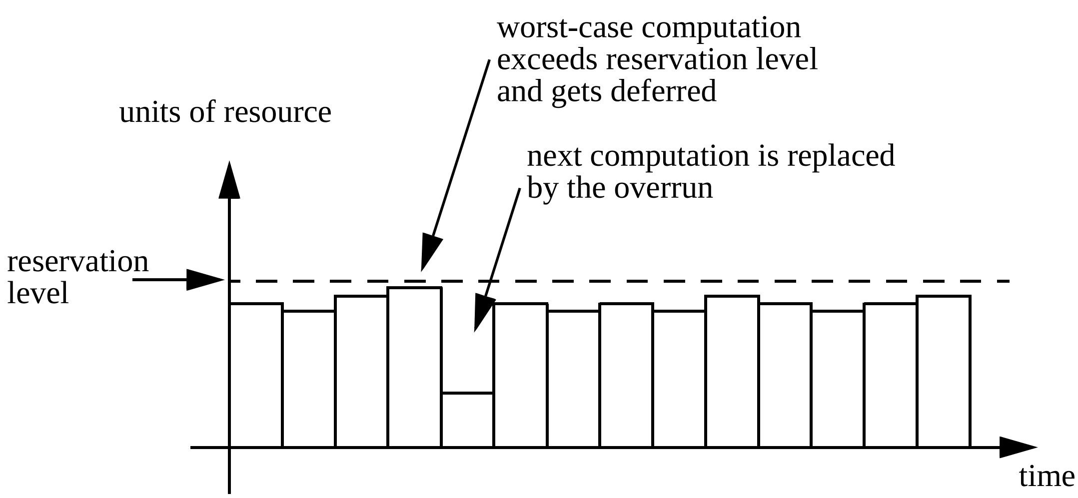

Figure 4-15: Computation Impinges on Following Computation

Figure 4-15 shows the case where the computation in the 4th reservation period extends into the next reservation period and prevents the next computation from being initiated. Computations following that are left undisturbed.

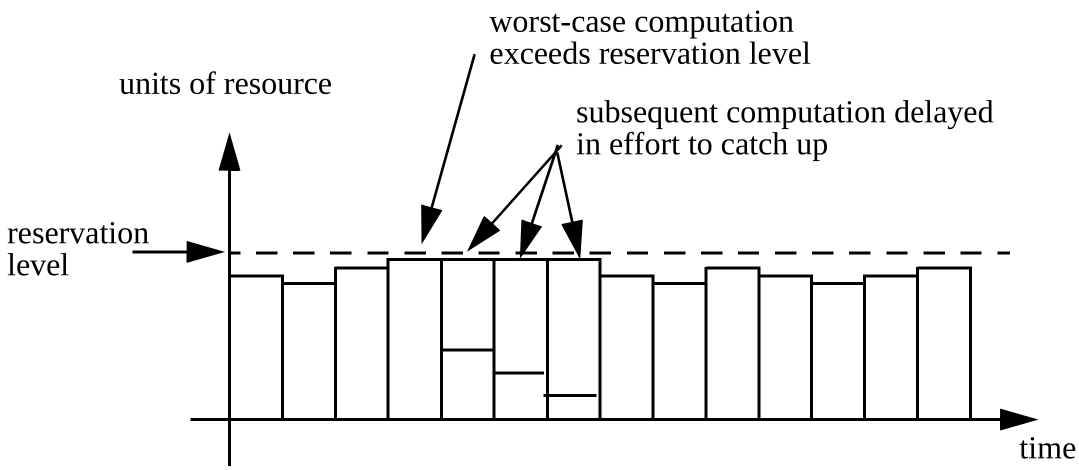

Figure 4-16: Computation Impinges on Subsequent Computations

In Figure 4-16 the 4th computation overruns its reservation period and consumes part of the next period. The computation associated with that period is initiated after the previous computation is completed (as opposed to the previous case where this computation was not initiated). But since the 5th computation is initiated later than usual, it also overruns its reservation period and is deferred to the next period. This cascading effect continues until there is enough (normally) unused but reserved units of resource to make up for the original overrun.

In cases where the average resource usage requirement is more than the reservation, the activity will never be able to accommodate all of the computations which overrun, and it would be necessary to shed some of the load by aborting some computations or by not initiating some computations. In either case, the attempted overruns would occur frequently and have a potentially damaging effect on overall application behavior. In a video player, for example, this would mean that many frames get dropped.

Figure 4-17 illustrates a possible pattern of demand that has the average demand greater than the reservation level. The computations in several reservation periods require more than the reservation for that period. Many of the computations will have to be aborted if there is no idle time available beyond the reserved level.

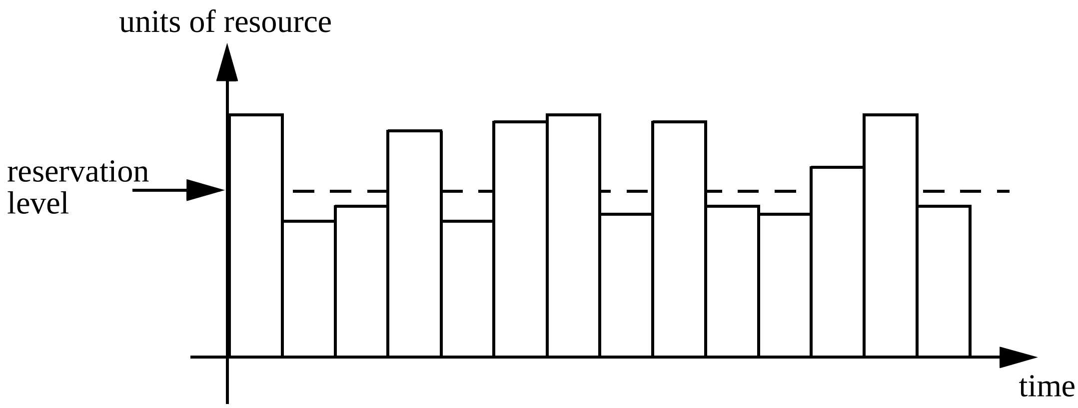

Figure 4-17: Average Demand Exceeds Reservation

4.3.3.3 Over-reserved applications

An application is over-reserved if the resource usage in each period is (much) smaller than the reservation level.

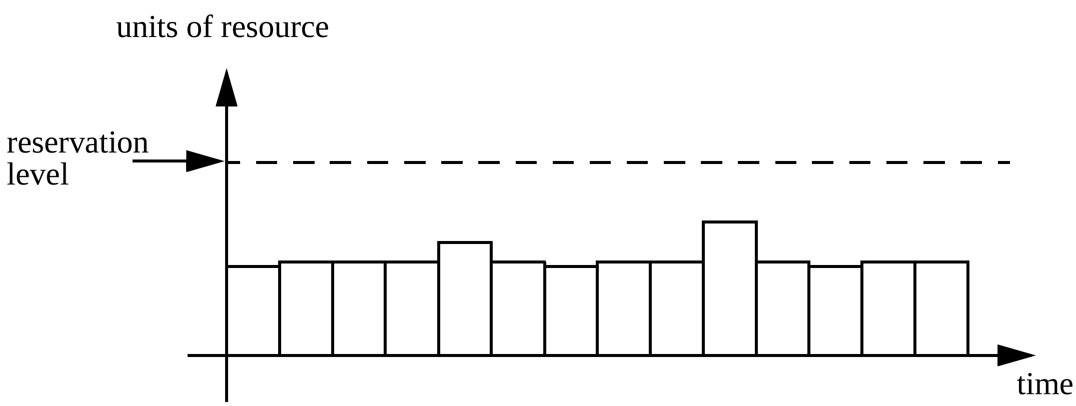

Figure 4-18: Resource Demand Smaller than Reservation

Figure 4-18 illustrates a case where the usage on a particular resource is much smaller than the reservation in all of the reservation periods. Here the computation is never in any danger of overrunning into the next period.

4.3.3.4 Multiple resources

When there are multiple resources involved in each computation, the measurements of usage compared to reservation level for each reserve will be different. One resource may be over-reserved while all of the others are under-reserved, or perhaps more commonly, one resource may be under-reserved (representing a bottleneck) while all of the other resources are over-reserved for the activity.

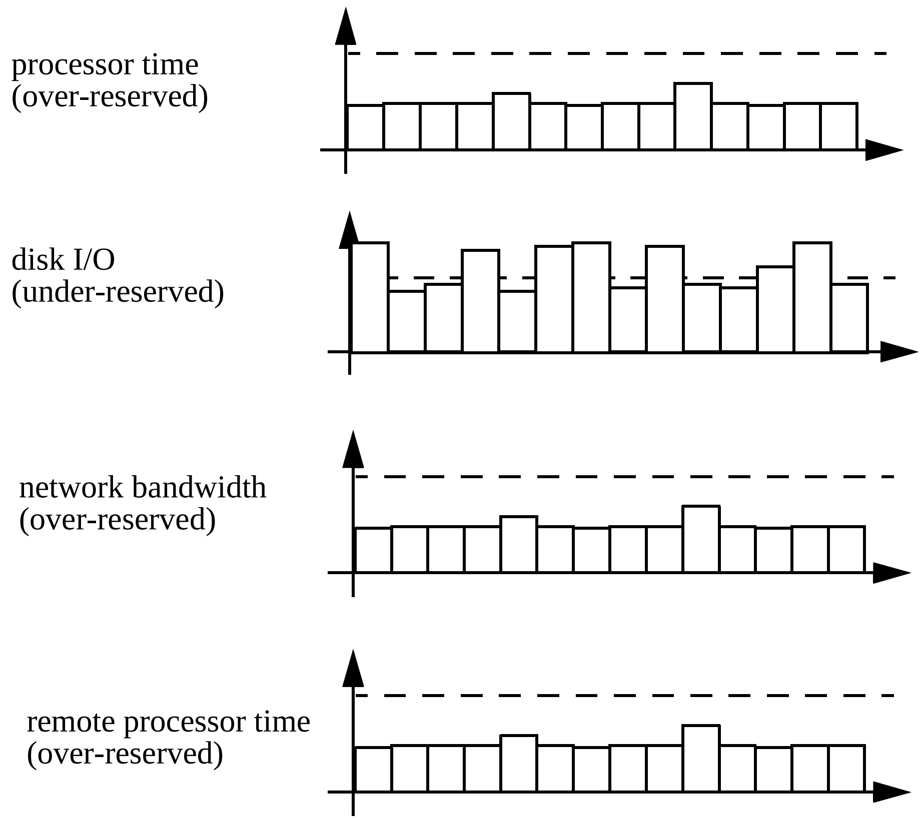

Figure 4-19: Measurements of Multiple Resources

Figure 4-19 illustrates a case where there are multiple resources involved in a single activity. They all have the same reservation period, but the demands placed on various resources are different. In this case, the disk I/O is under-reserved while the local processor usage, the network bandwidth, and the remote processor usage are all over-reserved.

Once an application has measurements of usage for the various resources it requires for its computation, it can begin to make decisions about how to modify its own behavior or modify its own resource reservation levels to achieve better performance or better efficiency.

4.3.4 Adapting

Reservation parameters can be changed dynamically as the user, the application itself, or a central quality of service manager determines that new reservation parameters would be preferable. Applications adapt based on the influences of various external entities, but once a resource reservation is made, the system ensures that the resources are available.

Adaptive applications can measure their own performance by mapping measured system resource performance metrics to application-specific performance indicators. This application-specific information can then be used along with application-specific performance objectives to:

1. modify the computation being done by the application (to change resource requirements),
2. modify the reservation level for resources being used by the application, or
3. do nothing.

The performance measurement interval could be comparable to the period of the repetitive computation, but it would be more efficient if the adaptation interval were an order of magnitude larger than the activity period (e.g. the adaptation might occur every 500 ms for an activity with a 50 ms period).

4.3.4.1 Modifying an application's computation

One way that an application might react to the fact that its resource usage is different from its reservation is to change its behavior so that its usage more closely matches its reservation (leaving the reservation unchanged). The actual mechanisms for modifying behavior in an application are fairly straightforward. An application which is meant to modify its own behavior must have different behaviors available (i.e. different algorithms implemented internally). It must be able to tell which algorithm it should use depending on the format of a incoming or outgoing data stream, on the resources such as network bandwidth or computational power that are available to it, or on the limitations of other software with which it must interact. For example, an MPEG video decoder could use different decoding or dithering algorithms depending on the resources available. If the decoder were taking the MPEG data stream from a server in real time, it might be able to negotiate MPEG encoding parameters with the server and have the server place new parameters in the data stream. Thus, adaptive applications are constrained by:

1. the algorithms they have coded,
2. the data formats they are using,
3. the data formats and data rates that other components of the pipeline can handle, and
4. the resources that are available to it.

Typically, the application would contain a collection of algorithms that could be ordered based on processor requirements, network bandwidth requirements, etc. Thus, once an adaptive application decided to increase or decrease a reservation on a particular resource, it could determine which algorithms could satisfy that constraint. It is important to distinguish between two kinds of behavioral adaptations:

1. local changes in algorithm and
2. global changes.

An example of a local change would be a change in the number of bits actually being decoded by the receiver of a bit stream (assuming the bit stream was encoded using a hierarchical encoding scheme). A global change would involve not only the receiver but the sender. It would require a way for the receiver to request a change in the format or number of bits being sent as well as requiring the receiver to recognize that the format of the bit stream changed.

As an example of what the code structure for an adaptive application would be, consider a video player. The basic control structure of the player is a loop that reads some data (from a disk, network or some other source), decodes the data, displays the data, and evaluates its performance.

```
while (1)
    read data for a frame
    decode data
    display frame
    evaluate resource usage
```

It is the evaluation part of this loop which will look at the resources that are being expended over time to play the video frames and decide whether the amount of work being performed should be increased, decreased, or remain the same. If it decides the work should be *locally* increased or decreased, it may change some state in the player to indicate how the data should be decoded (by looking at more or fewer bits of the data). If the evaluation phase decides that the work should be *globally* changed, it may initiate negotiations with the source of the data stream to try to increase or decrease the bandwidth of the bitstream. This negotiation may or may not change the state of the player itself, but the changes to the characteristics of the bit stream and the point in the bitstream where the change takes place should be clearly identified in the player and should be recognized in decoding the data. Thus, if and when a change in the format of the bitstream occurs, the player will be able to make the appropriate changes in decoding the stream.

So the software structure of the adaptive player, with a little more detail filled in, becomes:

```
while (1)
    read data
    check for control data
    switch based on bitstream format
        switch based on local decoding state
           decode data
    display frame
    evaluate performance
```

The application first reads the data and then checks either for control information embedded in the data or for some kind of synchronization point which is known, through information communicated via an external channel, to imply a change in data format. Then the proper algorithm is chosen to decode the data based on the format of the bitstream and on the local decoding state. Once the data is decoded, it is displayed (possibly using different algorithms indicated by the player state), and finally the performance is evaluated. To reduce overhead, the performance may not be evaluated during every iteration of the loop.

This example shows how a player might change its behavior and thus its performance characteristics based on decisions about local algorithms and global changes in data streams.

4.3.4.2 Adjusting reservation levels

Another way an application might react to noticing a difference between its usage and reservation is to change the reservation (without modifying its computation). We will examine several cases, possible behavior modifications, and their effects on delay, efficiency, and total reservation.

Under-reserved applications

As indicated in the section on measuring resource usage, an application is under-reserved if its resource usage requirements are greater than its reservation level. There are a couple of ways to change the reservation parameters to accommodate this situation:

1. increase only the units of reserved resource usage
2. increase both the units of reserved resource usage and the reservation period.

By increasing the reserved resource usage to match the computation's requirements, the application can ensure that the resources will be available to the computation, and the computation can be invoked just as often as before. The delay experienced by each computation will be decreased since there will be fewer overrun situations to cause delays, but the overall reservation level is increased. This means there is less resource capacity available for other applications, or when resource reservations are really tight, it may be impossible to increase the reservation level at all.

If the application increases the units of reserved resource and the reservation period proportionally, there will be enough reserved resource capacity in each reservation period to service the computation. And since the reserved amount and the reservation period are increased proportionally, the overall reservation level is not increased. This also implies that the computation is requested less often to correspond with the longer reservation period since requesting it just as often would not reduce the overall workload (without a load-shedding mechanism coming into play).

Over-reserved applications

Over-reserved applications are those which have a reservation level that is greater than the actual resource demand. This situation is inefficient since the application has more resource capacity reserved than it expects to use, and that reserved capacity could be used to ensure predictable performance for other applications that will actually use the resource.

The simplest action to take in this case is to reduce the units of resource reserved in each period to a value that is closer to the actual resource requirement. It is possible to increase the reservation period without increasing the period of the computations to decrease the reservation level, but that would have other undesirable effects on the timing of the program, such as increasing the delay for some computations.

4.4 Chapter summary

This chapter describes how programs should be structured to take advantage of reserves for predictable real-time performance. For hard real-time applications, information about the resources used by and timing requirements of each program must be known at design time and must be used in planning reserve allocation. So a localized way of using reserves might be appropriate. With local reserves, each program allocates its own reserves based on its requirements and the requirements of other programs that depend on it. For dynamic soft real-time systems, a global method for reserve allocation where an activity allocates the resources for all of its constituents including external servers and operating system services might be more appropriate. This makes it easier to monitor and control the usage of the entire activity rather than just localized parts of it. These recommendations are not cast in stone; the choice of whether to use localized or global reserve allocation ultimately rests with the system designer.

The discussion also dealt with methods for determining resources required by an application, reservation parameters appropriate for an application, and adaptive methods for adjusting reservation parameters or behavior based on performance history. In hard real-time systems, many of these questions must be answered at design time, and there is less flexibility in adaptation strategies. Soft real-time systems, however, have a great deal of flexibility and can take advantage of some of the techniques described in this chapter.