The Aura Software Architecture: an Infrastructure for Ubiquitous Computing

João Pedro Sousa, David Garlan

August 2003
CMU-CS-03-183

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Computing environments of the future should enable mobile users to take full advantage of the computing capabilities available at each location, while allowing them to focus on their real tasks, rather than being distracted by dealing with the configuration and reconfiguration of computer systems to support those tasks. The Aura infrastructure performs automatic configuration and reconfiguration of Ubicomp environments, according to the user's task and intent.

This report describes the software architecture of the Aura infrastructure, and discusses the underlying rational. It describes the architecture from a layered perspective, detailing the partition of responsibility and shared assumptions, as well as from a component-connector perspective, detailing the protocols of interaction between the components (APIs and sequencing). The contents and format of the exchanged messages is extensively discussed, as well as the details pertaining service interconnection and decomposition. This report proposes a utility-based approach for modeling user preferences, and details how such models can be exploited for both coarse-grain automatic (re)configuration, and fine-grain adaptation to resource change.

This material is based upon work supported by the National Science Foundation (NSF) under Grant CCR-0205266, and by DARPA under Grant DASA0001. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, DARPA, or Carnegie Mellon University.

Keywords: ubiquitous computing, software architecture, task-oriented computing, everyday computing, self-configurable systems, adaptive systems, model-based adaptation, modeling user preferences, utility-based adaptation, resource-adaptive applications.

Acknowledgments

The ideas contained in this technical report owe much to Mahadev Satyanarayanan and Peter Steenkiste for asking the right questions. Finding and polishing the answers to those questions emerged out of detailed discussions with Vahe Poladian, Bradley Schmerl, Rajesh Balan, and Dushyanth Narayanan. In addition, a number of others have contributed in various ways to the work underlying this report (in alphabetical order): Jason Flinn, Lalit Jina, Chiharu Kawatake, Mona Li, Takahide Matsutsuka, Tadashi Okoshi, Bhuricha Sethanandha, Chris Tuttle, Zhenyu Wang, Wei Zhang.

Our appreciation goes to Gaetano Borriello, with Intel Research Seattle, for providing a stimulating and nurturing environment for João P. Sousa, during the summer of 2002, where some of the ideas herein matured.

Contents

1 Introduction

Advances in technology are creating new expectations by users for capabilities delivered by emerging Ubiquitous Computing (Ubicomp) systems [10]. Increasingly, ordinary artifacts and physical spaces offer computing power to the end user: phones, entertainment systems, cars, airport lounges, cafés, etc. A natural consequence of this abundance is that people increasingly expect to push the use of computing beyond the desktop, scaling that use both in space and in time [1]. For instance, a user may start watching a video clip at home, and continue on the bus; he may join a teleconference while walking down the hall, and participate in it while sitting in a smart room; or he may be writing a conference paper on and off, in the free time between daily meetings and activities.

An important assumption of Ubicomp is that users want to take full advantage of the devices and resources available at each location. In contrast with the premises of Mobile Computing, Ubicomp users are not expected to carry private devices around, although they might.

However, taking full advantage of Ubicomp today comes at a cost for users. First, as users move from one location to the next, they must handle the chores of transferring their computer-supported tasks to the new *environment*.[1] Users have to deal with finding and configuring suitable components to support their tasks; and they have to deal with migrating/accessing the relevant information. Second, users are required to manage resources and dynamic change in the environment. To obtain the desired level of quality of service, users have to be aware of the demand that alternative computing modalities pose on limited resources, such as battery and bandwidth. And third, a setup that corresponds to the user's expectations at some point may be unacceptably poor a few moments later: for example, in heavily networked environments, remote servers constantly change their response times and even availability.

The broad problem being addressed by this work is how to increase the *utility* of Ubicomp environments for users. Casting the problem this way is motivated by the observation that with the increasing availability of environments rich in capabilities and resources, the focus of optimization should be on the user's experience. There are two aspects to the user's experience: *benefit* (how useful the environment is for the user's task) and *cost* (the user's overhead in setting up the environment for his task).

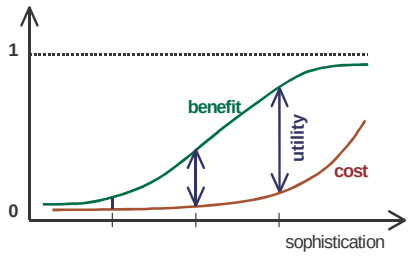

Figure 1. Cost/benefit of solutions against sophistication level

The *utility* offered to the user is defined as the difference *benefit – cost* (see Figure 1). Solutions offered in traditional environments typically have low cost for the user, but also offer low benefit. This is due to the inadequacy of such traditional solutions in addressing the characteristics of Ubicomp environments: heterogeneity, dynamic change, etc. However, increasing the level of sophistication arbitrarily does not necessarily lead to an increase in utility: adding more capabilities than the user can effectively apply in his task no longer adds to the benefit. Therefore, the

[1] Informally, the computing *environment* is the set of devices, applications and services that are accessible to a user standing at a particular location.

increase in benefit is limited and it eventually saturates. Often, sophisticated solutions require so much effort from the user to configure/train the system that the cost outweighs the increase in benefit. Ideally, the sophistication level of solutions hits the sweet spot: the point where the utility is maximized.

There are many subproblems within the broad goal of increasing the utility of Ubicomp environments for users, and many research avenues address those subproblems: natural interfaces, awareness of the user's physical context, non-intrusive learning of user tasks and intent, cognitive models of the user and the user's task, etc. One approach to increasing the utility is for the system to take over routine chores currently handled by the user.

The work reported herein focuses on the subproblem of defining an infrastructure for the *automatic configuration and reconfiguration of Ubicomp environments*, henceforth the *infrastructure* [8]. There are three aspects to this subproblem. First, as users move from one location to the next, the infrastructure automatically handles the chores of transferring their computer-supported tasks: finding and configuring suitable services to support their tasks, and dealing with migrating/accessing the relevant information. Second, as users switch from one task to another, or resume previous tasks, the infrastructure automatically sets up all of the relevant capabilities (*everyday computing* [1]). Third, the infrastructure shields the user as much as possible from distractions by automatically adapting to dynamically changing resources and capabilities [3].

To address the problem of automatic configuration and reconfiguration of Ubicomp environments, the infrastructure exploits lightweight models of user tasks. Such models capture the needs of the user in terms of required services in the environment, their interconnections, preferred characteristics, and levels of quality of service. To address environment diversity, user tasks are expressed in terms of abstract *services*, such as *text editing*, *video playing*, *printing*, etc.[2] By automatically searching, setting up and maintaining service configurations that best meet the user's needs, the *benefit* offered by the environment is increased for the duration of the user's task. By using lightweight models of user tasks, the *costs* are kept low for the user, specifically the overhead of learning the models of user tasks.

This report focuses on the software architecture of the Aura infrastructure. This work is part of Project Aura, a wider research thrust in Ubicomp at CMU [6]. Section 2 describes the layered view of the software architecture, and Section 3 describes the interaction model between the two top layers. Section 4 describes the component-connector view, including the protocols of interaction supported by the connectors, and Section 5 details the contents of the exchanged messages. Section 6 addresses interconnection of services, and Section 7 addresses service decomposition. Finally, Section 8 enumerates research questions relevant to this work, identifies which ones are addressed herein and points at future work.

2 Architectural layers

The Aura infrastructure exploits knowledge about the user's tasks to automatically configure Ubicomp environments on behalf of the user. For that, first, before any automatic configuration, the infrastructure needs to know *what* to configure for: what does the user need from the envi-

[2] The term *service* is often overloaded to mean (a) the service type, such as *printing*, (b) the occurrence of the service proper – printing a given document, and (c) a supplier of that service – a particular printer. For simplicity, I will let meanings (a) and (b) be inferred from context, and will consistently use the term *supplier* for meaning (c).

ronment in order to carry out his tasks. Second, the infrastructure needs to know *how* to best configure the environment: it needs mechanisms to optimally match the user's needs to the capabilities and resources in the environment.

In Aura, each of these two subproblems is addressed by a distinct software layer: (1) the Task Management layer determines *what* the user needs from the environment at a specific time and location; and (2) the Environment Management layer determines *how* to best configure the environment to support the user's needs.

layer	*mission*	*roles*
Task Management	***what*** does the user need	• monitor the user's task, context and intent • map the user's task to needs for services in the environment • map user intent to fidelity/resource tradeoffs • complex tasks: decomposition, plans, context dependencies
Environment Management	***how*** to best configure the environment	• monitor environment capabilities and resources • map service needs, and user-level state of tasks to environment-specific capabilities • ongoing optimization of the utility of the environment relative to the user's task
Environment	support the user's task	• monitor relevant resources • fine grain management of fidelity/resource tradeoffs

Figure 2. Summary of the software layers in the infrastructure

Figure 2 summarizes the roles of the software layers in the infrastructure. The top layer, Task Management, captures knowledge about user tasks and associated intent. Such knowledge is used to coordinate the configuration of the environment upon changes in the user's task or context. For instance, when the user enters a new environment, Task Management coordinates access to all the information related to the user's task, and negotiates task support with the Environment Management. Task Management also monitors explicit indications from the user and events in the physical context surrounding the user. Upon getting indication that the user intends to interrupt the current task or switch to a new task, Task Management coordinates saving the *user-level state* of the interrupted task and instantiates the intended new task, as appropriate (see Section 2.1). The Task Management layer may also capture complex representations of user tasks, including task decomposition (e.g., task A is composed of subtasks B and C), plans (e.g., C should be carried out after B), and context dependencies (e.g., the user can do B while sitting or walking, but not while driving).

The Environment Management layer holds abstract models of the environment. These models provide a level of indirection between the user's needs, expressed in environment-independent terms, and the concrete capabilities of each environment. This indirection is used to address both heterogeneity and dynamic change in the environments. With respect to heterogeneity, when the user needs a *service*, such as *speech synthesis*, the Environment Management will find and configure a *supplier* for that service among the components in the environment. With respect to dynamic change, the existence of explicit models of the capabilities in the environment enables automatic reasoning upon dynamic changes in those capabilities. The mapping between user needs and concrete applications/devices can thus be automatically adapted at runtime. In contrast, in traditional environments, where user mobility and dynamic change in the environment's capabilities are not an issue, typically this mapping is handled manually by the user. The Environment Management adjusts such mapping automatically, not only in response to changes in the

user's needs (adaptation initiated by the Task Management), but also in response to changes in the environment's capabilities (adaptation initiated by the Environment Management itself). In either case, adaptation is guided by the maximization of a *utility function* provided by the Task Management that captures user intent and preferences (see Section 3).

The Environment layer holds the applications and devices that can be *marshaled*, by the Environment Management, to support the user's task. Configuration issues aside, these components interact with the user in the same way as they would without the presence of the infrastructure. The infrastructure steps in only to the extent of automatically configuring those components on behalf of the user. The specific capabilities of each component are manipulated by the Environment Management Layer, which acts as a translator for the environment-independent descriptions of user needs issued by the Task Management.

The infrastructure can accommodate components with a wide range of sophistication in matters like fidelity- and context-awareness. By factoring models of user context and intent out of individual applications, the infrastructure makes it easier for each application to apply the tradeoffs and policies appropriate for each circumstance [2]. For instance, would the user of a language translator prefer accurate translations or snappy response times? Should an application running on a mobile device use power-save modes to preserve battery charge, or should it use resources liberally in order to complete the user's task before he runs off to board his plane? That knowledge is very hard to obtain at the application's level, but once it is determined at the user level – by the Task Management – it can easily be communicated to the applications selected to support the user's task.

Each layer reacts to changes in user tasks and in the environment at a different granularity and time-scale. Task Management acts at a human perceived time-scale (minutes), evaluating the adequacy of sets of services to support the user's task. The Environment Management acts at a time scale of a few seconds to evaluate the adequacy of the mapping between the requested services and specific components. Adaptive applications (fidelity-aware and context-aware) choose appropriate computation tactics at a time-scale of milliseconds.

2.1 User-level state

The user-level state of a task refers to the user-observable set of properties in the environment that characterize the support for that task. For example, the set of services marshaled to support the task, the user-level settings (preferences, options) associated with each of those services, the files being worked on, user-interaction parameters such as window size, cursors, etc. The ability to recover the user-level state of a task plays a fundamental role in the automatic configuration of the environment. Capturing and recovering the user-level state comes into play whenever the user moves from one environment to another (user mobility), when the user swaps one task for another (everyday computing), or upon failure or proactive reconfiguration of part, or all, of the environment supporting the user's task. The user-level state of a task is captured and recovered at the granularity of each service supporting the task.

The user-level state of each service is structured into three components: (a) the QoS preferences for the service, (b) the settings of the service, and (c) the materials used by the service. As a rule, QoS dimensions are (i) strongly correlated with resource demands, and (ii) the user accepts that their values fluctuate with resource availability. QoS preferences are used for tuning resource adaptation policies within the environment's resource constraints (see Section 3.1). In contrast, service settings may only be changed explicitly by the user during the operation of the service; and typically are not, or are weakly correlated with resource demands. Take, for instance, a *lan-*

guage translation service. The user is willing to let the infrastructure make automatic make tradeoffs on the (QoS dimensions) *latency* and *accuracy* of the translation, according to resource availability. However, the (setting) *languages object of translation*, for instance, English to Spanish, can only be set by the user.[3] Similarly, the user chooses explicitly which materials should be processed by a service, for instance which text document to edit. Additionally, the state of the materials is changed by the user during the operation of the service, although typically such change is implicit. For instance, typing some text changes the (material state) *cursor position* within the (material) document being edited.[4] See Sections 4.3 and 5 for a detailed discussion on how the user-level state is captured and recovered.

3 What-how interaction model

"What the user wants" is not always uniquely determined – and consequently it does not uniquely determine which services should be marshaled in the environment. For instance, the user may be willing to takes notes on a promotional video – but if the video cannot be played with adequate fidelity, maybe because of insufficient bandwidth, he may be willing to work on his weekly report instead. In everyday computing, users typically have several tasks they are willing to work on. Additionally, each task may have more than one way of being supported: for instance, for taking notes, the user may dictate, type, or write the text on a pad.

The set of services that ultimately should be marshaled in the environment – and how they should be configured – is derived by optimizing the match between what the user wants and what the environment has to offer. Therefore, the two top layers introduced in Section 2 cooperate in finding such match: the Task Management generates the alternatives for *what* the user may want, while the Environment Management evaluates *how* well each alternative can be supported by the environment. For that, the role of the Environment Management layer is to determine which are the components in the environment – and how to configure them – to best support user's task, given (a) a set of requested services, and (b) user preferences encoded in a suitable way. The role of Task Management is to determine item (b) above: what are the *real* user preferences *for the task at hand*? What levels of quality of service are acceptable? What tradeoffs is the user willing to make? There will be a fair amount of uncertainty associated with the answers to these questions, and to manage this uncertainty the infrastructure needs to learn the user's preferences in concrete situations. By observing the user's acceptance or refusal of the alternatives, the Task Management can tune its knowledge of the user preferences and improve its role as a user proxy for the configuration of the environment.

---

[3] Of course, settings are recovered by the Task Management layer, on the user's behalf, as part of the automatic configuration of the environment. However, settings cannot be set autonomously by the Environment Management layer, or by the applications themselves, as part of some resource adaptation policy.

[4] When designing the ontology for a service type, it is not always clear cut whether a concept should be a service setting or part of the state of a material. In such cases, the question to ask is: should the service work on more than one material simultaneously, do we need separate attributes for each material? The answer for *cursor position* in a text document is clearly yes; for *spell checking enabled* is probably not, although a richer model would also be acceptable.

3.1 User preferences

Computing the best match between what the user wants and what the environment has to offer corresponds to maximizing a *utility function*. The utility functions used in our work express formally, in a computable way, the user's preferences and intent for a specific task. This section describes a *utility framework*, consisting of a proposed *structure* for representing user preferences, and an efficient *strategy* to exploit such a structure for the automatic configuration and reconfiguration of Ubicomp environments. Section 3.2 describes the structure formally.

Structure of user preferences. User preferences (and their formal reification, utility functions) used in the Aura infrastructure have three parts: first, *configuration preferences* capture the preferences of the user with respect to the set of services to support a task. Second, *supplier preferences* capture which specific components are preferred to supply the required services; and third, *QoS preferences* capture the acceptable Quality of Service (QoS) levels and preferred tradeoffs.

As an example of configuration preferences, recall the task of reviewing a promotional video. For taking notes, the user may prefer to dictate the text. However, if the environment lacks the capabilities (microphone, speech recognition software…) or resources (CPU cycles, battery charge…) to support dictation satisfactorily, the user is willing to type or write the text. As another example, suppose the user is moving around carrying only his handheld, and he wants to watch a soccer game available from an on-line video feed. Since video and audio are competing for limited bandwidth, sometimes the video quality degrades so much that the user can no longer follow the game. When that happens, the user is willing to forego the video and have the meager bandwidth be allotted to provide acceptable audio. As an example of supplier preferences, for typing notes (*text editing* service), the user may prefer MSWord over Notepad or Emacs, and be unwilling to use the *vi* editor at all. As an example of QoS preferences, consider again watching the video for the soccer game over a network link. Suppose that the bandwidth suddenly drops: should the video player reduce image quality or frame-update rate? For the soccer game, frame-update rate should be preserved at the expense of image quality; however if the user were watching a painting critique, image quality should be preserved at the expense of frame-update rate. As another example of QoS tradeoff, if the user is using automatic translation and the resources are limited, would the user prefer more accurate translations or snappy response times?

Exploiting the structure of user preferences. Configuration preferences are used by the Task Management when deciding what to configure for the user. The Environment Management uses supplier preferences to make a first pass at finding the best candidates to support each requested service, and then it uses QoS preferences to make a final decision on which components are better positioned to deliver the QoS expected by the user. Finally, QoS preferences are used by the marshaled service suppliers themselves to determine appropriate resource-adaptation policies.

This utility framework is used to address the initial configuration, as well as for the ongoing reconfiguration of the environment to best support the user's task. Specifically, to address the initial configuration, the Task Management simply picks the alternative with the highest overall utility, as budgeted by the Environment Management. For the ongoing reconfiguration, different kinds of changes are addressed at different levels:

Task Management level. Changes in the user's task may be reflected in changing one or more services in the set currently in use. Since these are changes initiated at the user-level, the Environment Management evaluates them in the same way as it would evaluate an alternative in the initial configuration problem. In fact, the initial configuration of a new task is just an instance of this, where all the services need to be chosen. This kind of reconfiguration occurs at a human perceived time-scale (minutes).

Environment Management level. Reconfiguration is triggered whenever the marshaled set of components in the environment no longer offers the best utility for the requested set of services. Broadly, there are two causes for that: first, a change on the capabilities of the environment, such as new components becoming available or connected, or currently marshaled components failing or becoming disconnected. Second, a significant resource variation that causes the QoS offered by the currently marshaled components to drop below what is possible to achieve from other components. Whatever the cause, the user's task may be disrupted whenever the Environment Management initiates a reconfiguration that involves swapping one or more components. Whereas the user is expecting changes to occur in the configuration when his task changes (reconfiguration initiated by the Task Management at the user's direct or indirect request), he is not otherwise expecting changes in the environment (see **Cost of change**, below). The proactive reevaluation, and potential reconfiguration of the environment, is triggered by the Environment Management at a time-scale of a few seconds.

Environment level. Resource-adaptive applications can handle resource variations locally, according to the tradeoffs expressed in the QoS preferences. Such applications use QoS preferences to determine the appropriate computation strategies (i.e. adaptation policies), typically on a per operation basis: recognizing a speech utterance, rendering a virtual reality frame, or playing a video segment. QoS-aware suppliers gather statistics over time of the QoS actually being provided to the user, and thus enable the Environment Management level to periodically reevaluate whether a reconfiguration should be triggered.

Cost of change. The user is confronted with a cost of change whenever the Environment Management swaps a component supplying a service that involves direct interaction with the user. If nothing else, the user will be momentarily distracted from his task while shifting his attention to a new UI on the same device, or to a new device entirely. In such cases, even if the user welcomes the change, he may wish to make it at a convenient point in his work. For instance, if the component playing a video is about to be swapped, the user may wish to finish viewing the current scene; or if a component supporting taking notes is about to be swapped, the user may wish to finish his train of thought. For components that do not directly interact with the user the cost of change is typically smaller or negligible: the user may notice that the system staggers a bit while the components are being swapped in the background.

The utility framework incorporates the notion of cost of change associated with swapping a component at the level of the supplier preferences. The cost of change penalizes the utility offered by the new component, making the change attractive only when the utility is significantly better: the more the cost of change, the better the utility estimated for the new component has to be for the swap to be triggered by the Environment Management.

To account for the situations where the user may wish to delay the component swap to a convenient moment, the Environment Management issues a reconfiguration suggestion to the Task Management and waits for a confirmation. The Task Management may use knowledge about the user's task coupled with context observation to decide when the swap should take place; or it can default to prompt the user directly. Note that bringing that decision up to the user level effectively opens the possibility that the suggested swap is delayed indefinitely. From the point of view of the Environment Management, the protocols of interaction between Task and Environment Management must take into account that user intent is a moving target.

Consequently, the Environment Management does not block on the confirmation of the suggestion, but rather it continues to evaluate ensuing changes in the environment, possibly issuing new suggestions, and naturally it is open to distinct reconfigurations that follow from changes in the user's task. To reduce the chattiness of the interactions, the descriptions for the requested ser-

vices (not the utility function) state when the Environment Management has autonomy to swap a component for another with a higher utility (discounted by the cost of change). See Section 5 for the details of the message contents.

3.2 Formal underpinnings

This section describes the formal underpinnings of the user preferences discussed in Section 3.1. The QoS preferences for service *s* are expressed by:

$$\text{(QoS preferences)} \qquad U_{QoS}(s) \hat{=} \prod_{d \in QoS\ \dim(s)} F_d^{c_d}$$

where for each QoS dimension *d* of service *s*, $F_d : dom(d) \rightarrow (0,1]$ is a function that takes a fidelity value in the domain of *d*, and the exponent $c_d \in [0,1]$ reflects how much the user cares about QoS dimension *d*. As an example, suppose that the *play video* service includes a QoS dimension for *frame update rate*. The function $F_{frameRate}$ will be close to 1 for frame update rates that the user is happy with, and close to 0 for rates the user is unhappy with. Notice that the more sensitive the user is to variations in frame rate, the closer to 1 the exponent $c_{frameRate}$ is, and that $c_{frameRate}=0$ when the user doesn't care about such variations at all. The QoS utility of a supplier for *s* is obtained by applying U_{QoS} above to the specific fidelity levels being offered.

QoS preferences are encoded using a vocabulary of functions shared among the Task and Environment Management layers. The question becomes which vocabulary of functions *F* to choose in a continuum between the generic mathematical functions, such as multiplication, exponentiation, etc., and a reduced set of higher-level functions. The lower end of the spectrum has the advantage of being generic: it can support any function that can be encoded in standard mathematics and interpreted by programming languages. However it has two strong disadvantages: first, the implementation issues of parsing and evaluating the functions are harder the more generic the vocabulary; second, and most importantly, it is very hard to learn an arbitrary function that represents user preferences. Choosing a restricted set of high-level functions makes both these aspects easier, but it exposes the research question of choosing an appropriate vocabulary.

To make QoS preferences easier to both process and elicit, we make two simplifying assumptions with respect to their form. First, the preferences for each QoS dimension are modeled independently of each other. In other words, the preference function for each quality dimension captures the user's preferences for that dimension independently of other dimensions. Second, for each continuous QoS dimension, we characterize two intervals: one where the user considers the quantity as good-enough for his task, the other where the user considers it insufficient.

Sigmoid functions characterize such intervals and provide a smooth interpolation between the limits of those intervals (see Figure 3). Sigmoids are easily encoded by just two points: the values corresponding to the knees of the curve; that is, the limits *good* of the good-enough interval, and *bad* of the insufficient interval. The case of when less-is-better (e.g. latency) is just as easily captured as the case where more-is-better (e.g. image quality, as in Figure 3) by flipping the order of the *good* and *bad* values. [5]

[5] In practice, an adequate approximation for our purposes is given by a piece-wise linear function where $F(x)$ is an arbitrarily small positive constant for $x \leq bad$, $F(x)=1$ for $x \geq good$, and follows a linear interpolation between those two points.

$$F(x) = \frac{1}{1 + e^{\frac{good+bad-2x}{good-bad}}}$$

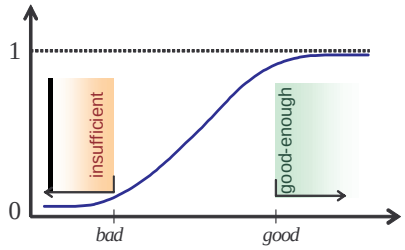

Figure 3. Generic sigmoid shape

For discrete QoS dimensions, for instance audio fidelity, with values *high*, *medium* and *low*, we simply use a discrete mapping (table) to the utility interval [0,1]. In the case studies we evaluated so far, we found the expressiveness of the forms above to be satisfactory.

The utility of the supplier assignment for a set *a* of requested services is:

(supplier preferences) $$U_{Supp}(a) \hat{=} F_w^{c_w} \cdot \prod_{s \in a} h_s^{x_s} \cdot F_s^{c_s}$$

where $F_w : time \rightarrow (0,1]$ is a function that takes the warm-up time and reflects how much the user is willing to wait for the suppliers to be set up; the exponent $c_w \in [0,1]$ reflects how much the user cares about the warm-up time; for each service *s* in the set *a*, $F_s : Supp(s) \rightarrow (0,1]$ is a function that appraises the choice for the supplier for *s*; and the exponent $c_s \in [0,1]$ reflects how much the user cares about the assignment of a particular supplier type for that service. Note that discriminating the *supplier type*, e.g. a preference of MSWord over Notepad for the *text editing* service, is a compact representation for the preferences with respect to the availability of desired features, such as spell checking or richness of editing capabilities, and to the user's familiarity with the way those features are offered. Naturally, all the F_s are discrete mappings.

The overall warm-up time of each supplier assignment is calculated by the Environment Management, and F_w will penalize the supplier assignments with warm-ups that the user perceives as long. A typical form for F_w is a sigmoid on the warm-up time. The term $h_s \in (0,1]$ reflects how (un)happy the user will be if the supplier for service *s* is exchanged during use: a value close to 1 means that the user is fine with the change, the closer the value is to zero, the less happy the user will be. The exponent x_s indicates whether such change penalty should be considered (x_s=1 if the supplier for *s* is being exchanged by virtue of dynamic change in the environment) or not (x_s=0 if the supplier is being newly added or replaced at the user's request).

The utility of the possible alternatives to support a task *t* is:

(configuration preferences) $$h_a \in [0,1]$$

where $a \in config(t)$ is a set of services than can support the user's task *t*, and the term h_a reflects how happy the user is with the alternative *a*. As an example, for the task of reviewing a promotional video, the user may signal that he is happy dictating the notes, ok with writing them, and will not accept typing them, by setting $h_{video\&dictate}$=1, $h_{video\&write}$=0.5 and $h_{video\&type}$=0.

Given a set of requested services *a*, and preferences U_{QoS} and U_{Supp} as above, the Environment Management determines the optimal supplier assignment $\hat{p}_s$, and the fidelity points $\hat{f}_d$ at which each supplier should run, by finding:

(EM) $$\underset{\substack{p_s \in Supp(s) \\ f_d \in dom(d)}}{\arg\max} \; F_w^{c_w}\left(\max_{s \in a} W(p_s)\right) \cdot \prod_{s \in a} h_s^{x_s} \cdot F_s^{c_s}(p_s) \cdot \left(\prod_{d \in QoS\,\dim(s)} F_d^{c_d}(f_d) \right) \Bigg|_{resources,\, QoSprof(p_s)}$$

constrained by the QoS profile of each possible supplier p_s, and by the available resources in the environment. Note that under the assumption that the suppliers to be added can be activated in parallel, the overall warm-up time is the maximum of the warm-ups for each supplier $W(p_s)$. The utility returned to the Task Management is the estimated utility offered by the optimal set of suppliers once they are up and running (without the penalty for supplier exchange or warm-up time):

$$\text{(budget)} \qquad \hat{U}(a) = \prod_{s \in a} F_s^{c_s}(\hat{p}_s) \cdot \prod_{d \in QoS\,\dim(s)} F_d^{c_d}(\hat{f}_d)$$

The Task Management decides which set of services *a* should be ultimately marshaled from the environment to support task *t* by maximizing:

$$\text{(TM)} \qquad \arg\max_{a \in config(t)} \; h_a \cdot \hat{U}(a)$$

Periodically, the Environment Management evaluates the utility of the configuration at the fidelity $\bar{f}_d$ actually being provided by the marshaled set of suppliers $\bar{p}_s$, and compares it with the utility of what would be the currently optimal configuration, $\hat{p}_s$ running at $\hat{f}_d$, discounted by the cost of change and warm-up time. Specifically, the Environment Management will consider a reconfiguration if the inequality below evaluates to true:

$$\prod_{s \in a} F_s^{c_s}(\bar{p}_s) \cdot \left(\prod_{d \in QoS\,\dim(s)} F_d^{c_d}(\bar{f}_d) \right) < F_w^{c_w} \left(\max_{s \in a|_{x_s=1}} W(p_s) \right) \cdot \prod_{s \in a} h_s^{x_s} \cdot F_s^{c_s}(\hat{p}_s) \cdot \left(\prod_{d \in QoS\,\dim(s)} F_d^{c_d}(\hat{f}_d) \right) \Bigg|_{\substack{resources, \\ QoSprof(\hat{p}_s)}}$$

4 Connectors & Components

The previous sections described the layered view of the Aura infrastructure. Specifically, we discussed the responsibilities of the two proposed layers, Task Management and Environment Management, and presented a framework for representing and exploiting user preferences in the context of automatic (re)configuration.

In this section we describe the components and connectors of the infrastructure, and we formally specify the protocols of interaction between such components. For the sake of research scope, we focus on the case of a single user; that is, we will not consider issues of coordinating the tasks of multiple users, or of reconciling conflicting user preferences. Additionally, we assume that at a given time and location, the user interacts with a single instance of the infrastructure.[6]

Figure 4 shows a component-connector type view of the infrastructure superimposed on the layers introduced in Figure 2. There are four component types: first, the *Task Manager*, called *Prism*, acts as a user proxy, coordinating the automatic configuration and reconfiguration of the environment. Second, the *Context Observer* provides information on the physical context surrounding the user, and reports relevant events in the physical context back to Prism, the Environment Manager, and context-aware applications. Third, the *Environment Manager* (EM) offers the mechanisms to marshal the supply of services required by user tasks. And fourth, *Suppliers* provide the abstract services that tasks are composed of: *text editing*, *video playing*, etc.

[6] Suppose that, for autonomy purposes, the user carries around a laptop with an instance of the infrastructure. When the user enters a location containing another instance of the infrastructure, say his office, presumably the user will expect the two infrastructures to cooperate so that he can access all the capabilities seamlessly. For the time being, and for simplicity sake, we are not addressing such cooperation.

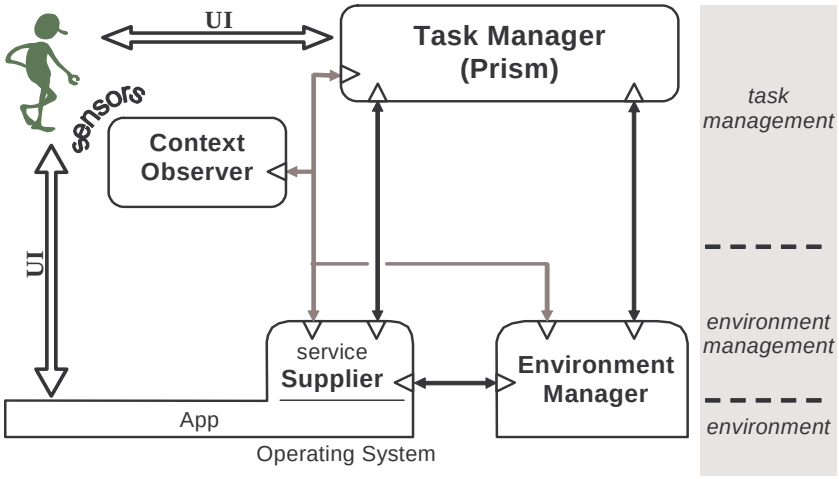

Figure 4. Component-Connector Type view of the infrastructure

From a logical standpoint, an environment has one instance of each of the types: Prism, EM, and Context Observer. Although the boundaries of an environment are defined administratively, they typically correspond to some physical area, like a floor or a building. Each environment typically has many service Suppliers: the more it has, the richer the environment is.

In practice, Suppliers are implemented by wrapping existing applications to conform to the infrastructure's APIs. Rather than requiring writing a new portfolio of applications, this approach makes it easy to integrate legacy applications into the Aura infrastructure. For instance Emacs, MSWord and Notepad can each be wrapped to become a supplier of *text editing* services.

While the EM contains generic mechanisms for identifying and marshalling Suppliers, the Suppliers encapsulate knowledge that is domain-, application-, device-, and (architectural) style-specific. Suppliers need to be aware of the specifics of the application's APIs, communications infrastructure, etc. A Supplier acts as a translator between the generic configuration directives issued by the EM and Prism, and the specific configuration APIs offered by the component it encapsulates. In other words, a Supplier knows *how* to configure the component, given a description of *what* the user needs. More generally, Suppliers encapsulate the knowledge on *how* each particular service can be obtained from the environment, whether from a single component, or from an assembly of several components – see service decomposition in Section 7.

We assume that the EM has no means of controlling the resources spent by a Supplier other than by setting constraints on the Supplier, monitoring the resources actually spent, and replacing the Supplier if it refuses to comply with the resource constraints. When calculating the fidelity points that optimize the overall utility, the EM may reach a solution that favors some services over others. Consequently, in order to achieve optimal utility, resources may be distributed unevenly among services. It is the Supplier's responsibility to keep its resource demands within the bounds set by the EM by adjusting the computation within the component it wraps.

In the remainder of this section, we first discuss how some requirements imposed by Ubicomp environments are addressed at the level of the connectors. Following, we describe the protocols of interaction supported by the connectors shown in solid black in Figure 4, as well as the effect of such interactions in the state kept by the EM. A separate report will describe the protocols involving the Context Observer.

4.1 Fiber of the connectors

The connectors between the components shown in Figure 4 play an important role in addressing the characteristics of Ubicomp environments. Typical Ubicomp environments are heavily distributed – the Suppliers, especially, may be scattered across different devices, some of which may be remote to the user's location. Connectivity varies widely, from high-speed wired connections to fluctuating wireless (radio or infrared) connections. Moreover, heterogeneity of devices and software components is a given. Even among the components of the infrastructure, each has to be ready to communicate with different versions of the others. These characteristics impose constraints on the communication style supported by the connectors. Specifically, this section discusses constraints with respect to the synchronicity of communication and to the format of the exchanged information.

Asynchronous, peer-to-peer communication. In synchronous communication, the originating (calling) component blocks on the reply of the target (called) component. However, in Aura, each component should keep up with its responsibilities in real-time, doing the best it can with the available information, and without blocking on another component's reply. For example, the EM should not stop monitoring the capabilities of the environment, or replying to Prism's requests, on account of being blocked on the reply of a remote supplier – which may have become disconnected. Likewise, Prism should not stop responding to changes in the user's task, when waiting for the reply of some other component.

Therefore, all communication between Prism, the Context Observer, the EM, and the Suppliers is asynchronous (non-blocking). By the same token, when any of the components generates a piece of information that is relevant for the others, it has the ability to communicate it immediately without having to wait for a request. For instance, when Prism first wants to instantiate a task, it will request the EM to find the best match of capabilities in the environment. However, if later the environment changes in a way that justifies a reconfiguration, the EM takes the initiative of coming back to Prism suggesting the reconfiguration, or just informing Prism that it performed the reconfiguration.

XML-based tagged format. Tagged formats have the advantage over raw data formats that they make it easier to deal with heterogeneity. Specifically, tagged descriptions of user tasks can be processed by components with different degrees of sophistication. For example, suppose the user requires a *text editing* service, and would prefer spell checking to be activated. Although finding a suitable Supplier in a rich environment will not be a problem, a basic text editor on a small platform may not support spell checking, or even be aware of what "spell checking" means. Therefore, the description of the required service must be such that a given Supplier is able to extract the information it can recognize, without being thrown off by information it does not know how to handle.

A prerequisite for tagged formats to address this problem is that Suppliers of a given service type share a vocabulary of tags and the corresponding interpretation. In the example, a tag for the "spell checking" feature would have to be agreed upon, and hold an unambiguous meaning. Naturally, each service type is characterized by a distinct vocabulary of tags corresponding to the information relevant for the service, although there may exist commonalities across service types. A similar argument applies to all communication between Prism, the EM, etc. Given the dynamic nature of these environments, any component may have to deal with different versions of the other components at some point in time, therefore all communication among the components shown in Figure 4 is XML-based.

The remainder of this section describes the protocols of interaction between Prism and the EM, between Prism and Suppliers, and between the EM and Suppliers – and how those interactions affect the state kept by the EM [9].

4.2 Prism – EM Protocol

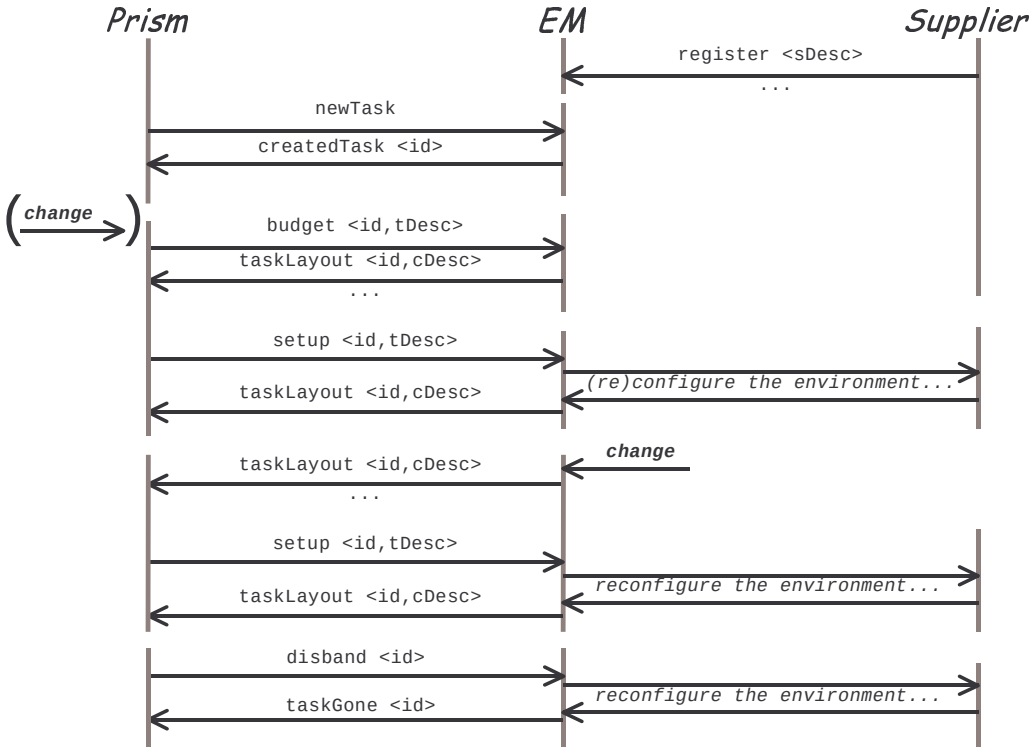

Figure 5. Event sequence diagram for the communication between Prism and the EM

The interaction between Prism and the EM is structured around the notion of *task session*. Prism initiates a session for task A whenever the user starts working on task A, and closes the session whenever the user interrupts or finishes working on that task. Figure 5 shows an event sequence diagram that illustrates a typical task session.[7] Prism starts a task session by sending the EM a `newTask` message. The EM reply, `createdTask`, includes an id for the task session that will be attached to the exchanged messages throughout the session. Note that many sessions between Prism and the EM may be active concurrently, for one or more users. The session is terminated by a `disband` message issued by Prism, to which the EM replies with a `taskGone` message.

Once a session is started, Prism obtains estimates for the utility that the environment can offer for the user's task by sending `budget` messages to the EM. Since there may be alternative configurations of services to support a given task, Prism will send one `budget` message for each of those candidate configurations. For example, for taking notes, either *text editing* or *speech recognition* services may be used. To each `budget` request, the EM replies with a `taskLayout`. For each service requested within a `budget`, the `taskLayout` indicates the Supplier that best matches the

[7] The interactions between the EM and Suppliers are elided for simplicity. Note that Suppliers register a description of the services they offer with the EM. More on this in Section 4.5.

request among those currently available in the environment, as well as the Quality of Service (QoS) achievable with the current resources. The `taskLayout` also indicates the overall utility for the configuration (see Section 5 for details). A budgeting exchange is also initiated by Prism whenever there are changes in the user's context or intent that justify a reevaluation of alternative configurations for the same task.

After evaluating the candidate configurations against the environment's capabilities, Prism decides (possibly with user involvement) which services to set up, and issues a corresponding `setup` message to the EM. Once the services are set up in the environment (see Section 4.5), the EM replies with a `taskLayout` containing the up-to-date description of the configuration supporting those services. Note that since there is a time lag between a `budget` and the `setup`, there may be some differences in what is achievable in the environment. Ideally, the contents of the two corresponding `taskLayout` messages will be the same, but Prism must be prepared to double check that, and either accept the differences, or if they are too significant, look for other alternatives and request a reconfiguration.

Of course, the capabilities of the environment may change, for better or for worse, during a task session. For example, a Supplier involved in the activated configuration may fail or become disconnected; a new Supplier that is a better match for a requested service may become available; or the resource conditions may change so much, that a different choice of Suppliers with distinct resource demands may be preferable. It is up to the EM to monitor the environment and promptly detect such situations. The EM will then reexamine the best match for the requested services, and it will either carry out the reconfiguration autonomously, or issue a `taskLayout` message to Prism containing a reconfiguration suggestion (establishing the policies for taking this decision is discussed in Section 3.1). Upon receiving a `taskLayout`, Prism may wish to study other alternatives to support the task, which it can do by issuing `budget` messages, but it will eventually settle on a reconfiguration and send the corresponding `setup` message.

Figure 6 shows the FSP specification for the protocol of interaction between Prism and the EM (Finite State Processes, FSP, is a process algebra akin to Hoare's CSP [7]). The permissible sequence of messages exchanged during a task session is specified by the `TaskSession` process.[8] After a `createdTask`, the protocol accepts either a `budget` or `setup`, leading to the `AnswerReq` process, a `taskLayout` initiated by the EM, or a `disband`, leading to the `DisbandSess` process. In the case Prism initiates a `budget` or `setup`, the EM is expected to reply with a `taskLayout` message. In the case Prism initiates a `disband` request, the EM is expected to reply with a `taskGone` confirmation. In both these cases, if the EM fails to reply, the protocol will engage in the `noEMreply` event, leading to the `RestartEM` process. Of course, the `noEMreply` event does not correspond to a real message, but rather to a timeout within Prism leading to a state change in the protocol of interaction. Similarly, the `resetEM` event does not correspond to a message, but to Prism activating a mechanism for rebooting the EM. Notice that after a `taskGone` message, the protocol goes back to the initial state awaiting the start of a new session with the `createdTask` message.[9] The `CreateSess` process states that after a `newTask` message is initiated by Prism, the EM is expected to reply with a `createdTask` message indicating the id for the session.

[8] Note that in FSP, the originator of each message (or *event*, in process-algebra terms) is unspecified, so a trace permitted by this specification should be understood as a possible sequence of messages observed in the channel between Prism and the EM, with the direction of communication abstracted away.

[9] In FSP, processes are not dynamically created and terminated, but rather transition back and forth from an active state to an inactive state, where all they can accept is the event corresponding to the "creation."

```
TaskSession = ( createdTask    -> MoreTaskReq ),
MoreTaskReq = ( {budget,setup} -> AnswerReq
              | taskLayout     -> MoreTaskReq     // EM's initiative
              | disband        -> DisbandSess ),
AnswerReq   = ( taskLayout     -> MoreTaskReq
              | noEMreply      -> RestartEM ),
DisbandSess = ( taskGone       -> TaskSession
              | noEMreply      -> RestartEM ),
RestartEM   = ( resetEM        -> MoreTaskReq ).

CreateSess  = ( newTask -> t[id:TId].createdTask -> CreateTask ).

||PrismEM   = ( CreateSess
             || forall [id:TId] t[id]:TaskSession )
             /  { noEMreply / {t[TId]}.noEMreply,
                  resetEM   / {t[TId]}.resetEM }.
```

Figure 6. FSP specification for the connector Prism-EM

The Prism-EM protocol is given by the parallel composition of the process for creating new sessions, `CreateSess`, and of some arbitrary number of processes of type `TaskSession`. In the FSP specification, this arbitrary number of processes of the same type is achieved by prefixing the process (and consequently the events within that process) by a label (`t`) and a number (`id`, in the arbitrary range `TId`). The process `CreateSess` and each specific `t[id].TaskSession` process interact by sharing the event `t[id].createdTask` – this models a new task session being created and named. Furthermore, all `t[id].TaskSession` processes share the `noEMreply` and `resetEM` events making sure all task sessions agree on when the EM needs to be restarted. This synchronization in the process model is achieved by relabeling all `t[id].noEMreply` events to a single event `noEMreply`, and the same for `resetEM`.

4.3 Prism – Suppliers Protocol

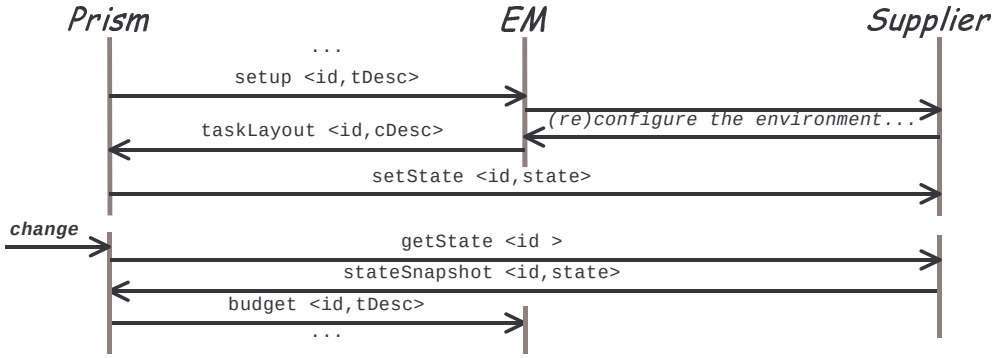

Figure 7. Event sequence diagram for the communication between Prism and the Suppliers

Figure 7 shows an event sequence diagram that illustrates a typical interaction between Prism and a Supplier. After Prism receives confirmation from the EM that Suppliers have been activated to support the user's task, Prism reconstitutes the user-level state of the task by sending a `setState` message to each of the Suppliers. Examples of user-level state are which files the user is working on, as well as user-interaction parameters such as cursors, window size, etc. (see Section 2.1). Prism recaptures the updated state from the Suppliers by sending a `getState` message to each, and receiving back a `stateSnapshot`. Recapturing the state of the Suppliers is done whenever there are changes in the user's context or intent that hint that the task is about to be suspended. It

may also be done periodically, to ensure recovery of an almost up-to-date state in the case a Supplier fails.

```
||PrismSupplier = forall [id:TId] t[id]:SetGetState.
SetGetState  = ( setState -> SetGetState
               | getState -> ( stateSnapshot -> SetGetState
                             | noSuppReply   -> SetGetState )).
```

Figure 8. FSP specification for the connector Prism-Suppliers

Figure 8 shows the FSP specification for the protocol of interaction between Prism and the Suppliers. For each task session, the protocol admits any sequence of `setState` and `getState` messages, with the proviso that a `stateSnapshot` reply is expected after each `getState`. Similarly to the `noEMreply` event in the Prism-EM connector, `noSuppReply` corresponds to a timeout within Prism rather than to an exchanged message. In this case, however, Prism will not take any action to recover/restart the Supplier. Prism relies on the EM to diagnose and propose the replacement of faulty Suppliers. Therefore, Prism will wait for some indication from the EM, or otherwise try to get the state again. The following section addresses combining these two protocols within Prism.

4.4 Prism

```
||Prism     = ( InvokeTask
              || forall [id:TId] t[id]:Task
              || forall [id:TId] t[id]:UseSupp )
              /  { noEMreply / {t[TId]}.noEMreply,
                   resetEM   / {t[TId]}.resetEM }.

InvokeTask  = ( newTask -> t[id:TId].createdTask -> InvokeTask ).

Task        = ( createdTask    -> CreatedTask ),
CreatedTask = ( {budget,setup} -> GetLayout
              | taskLayout     -> CreatedTask     // EM's initiative
              | disband        -> DisbandTask ),
GetLayout   = ( taskLayout     -> CreatedTask
              | noEMreply      -> RestartEM ),
DisbandTask = ( taskGone       -> Task
              | noEMreply      -> RestartEM ),
RestartEM   = ( resetEM        -> CreatedTask ).

UseSupp     = ( setup -> SetState
              | {taskLayout,noEMreply,disband}  -> UseSupp ),
SetState    = ( taskLayout -> setState -> SetGetState
              | noEMreply  -> UseSupp ),
SetGetState = ( {setState,getState,noEMreply} -> SetGetState
              | taskLayout -> ( replaceSupplier -> UseSupp
                              | keepSupplier    -> SetGetState )
              | setup      -> SetState
              | disband    -> UseSupp )
             \ {keepSupplier,replaceSupplier}.
```

Figure 9. FSP specification for the behavior of Prism

Figure 9 shows the FSP specification for the behavior of Prism. That behavior is the parallel composition of the process for creating new sessions, `InvokeTask`, of an arbitrary number of task session interactions with the EM, `Task`, and of the same number of interactions with Suppliers, `UseSupp`. The processes `InvokeTask` and `Task` state Prism's view of the processes `CreateSess` and `TaskSession`, respectively, in the Prism-EM protocols (they are restated here

for completeness of the specification of Prism's behavior). Notice also that, as before, the events `noEMreply` and `resetEM` are shared among the processes for all task sessions.

The glue between the two protocols Prism-EM and Prism-Supplier is specified by the `UseSupp` process. For simplicity, the messages exchanged with all the Suppliers supporting one user's task are modeled as single events. For instance, the `setState` event corresponds to sending `setState` messages to all such Suppliers. The central rule governing this protocol is that after receiving a `taskLayout` in response to a `setup`, the state of the Suppliers must be set. Therefore, initially, the `UseSupp` process looks for `setup` events and is permissive to other messages – for instance, `taskLayout` may occur is response to a budget. After the `setup` event, the `SetState` process looks for the corresponding `taskLayout`, after which it issues a `setSate`, or in case the EM fails to respond, it resets to `UseSupp`. After the initial `setState`, the `SetGetState` process allows Prism to issue any number of `setState` and `getState` messages. However, if a `taskLayout` is received at this stage, it will correspond to the EM's initiative to substitute a Supplier. In such a case, Prism must decide whether to keep the supplier or to request a replacement by issuing a `setup` (in `UseSupp`). Notice that both `keepSuplier` and `replaceSupplier` are local events just for the purpose of modeling Prism's decision. Additionally, as a consequence of a change in user intent, Prism may decide to request a change in the configuration by issuing a `setup` and waiting for the corresponding `taskLayout` (in `SetState`). Naturally, disbanding the task session resets the process to its initial state.

4.5 Protocol EM – Suppliers

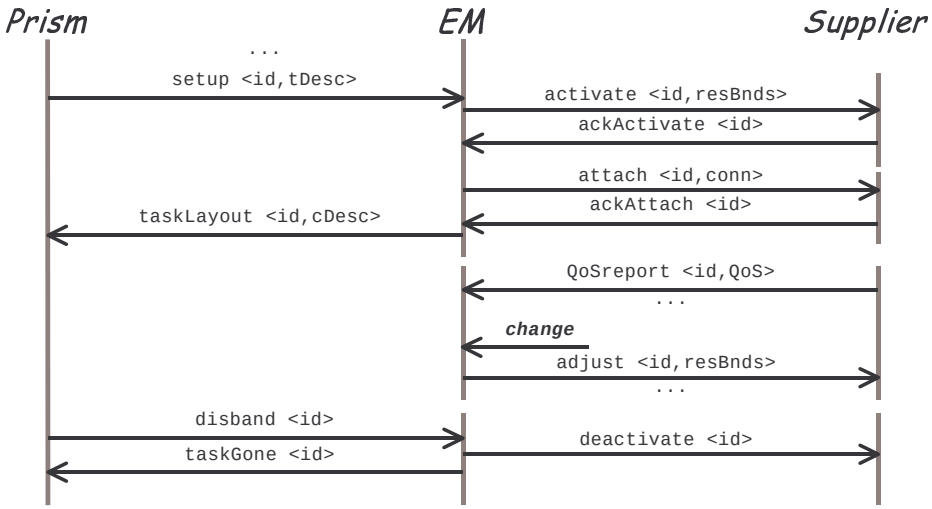

Figure 10. Event sequence diagram for the communication between the EM and the Suppliers

Figure 10 shows an event sequence diagram that illustrates a typical interaction between the EM and a Supplier. After the EM receives a `setup` request from Prism, it activates the Supplier, indicating the bounds on resource consumption, and it attaches the Supplier's ports as requested in the `setup` message. Both the `activate` and `attach` messages are acknowledged by the Supplier upon successful completion. After the supplier is activated, it issues periodic QoS reports to the EM. If the resources in the environment change significantly, the EM may establish new resource bounds for the Supplier by sending it an `adjust` message with the new bounds. Eventually, the EM will receive a disband message from Prism and proceeds to deactivate the Supplier. Notice that the EM makes sure the Supplier is up and properly attached before return-

ing a `taskLayout` to Prism. Subsequent adjustments to resource bounds and deactivation are not subject to the same constraint, and therefore, acknowledgments are not required.

```
||EMSupplier = forall [id:TId] t[id]:ManageSupp.

ManageSupp   = ( activate        -> ActivateSupp
               | deactivate      -> ManageSupp ),
ActivateSupp = ( ackActivate     -> AttachSupp
               | noSupplierReply -> ManageSupp ),
AttachSupp   = ( attach   -> ( ackAttach       -> MonitorSupp
                             | noSupplierReply -> ManageSupp )
               | noAttach                      -> MonitorSupp ),
MonitorSupp  = ( pQoSreport -> MonitorSupp
               | adjust     -> MonitorSupp
               | activate   -> ActivateSupp      // other Services
               | deactivate -> ManageSupp )
              \ {noAttach}.
```

Figure 11. FSP specification for the connector EM-Suppliers

Figure 11 shows the FSP specification for the protocol of interaction between the EM and the Suppliers. For each task session, the protocol is given by the process `ManageSupp`. Again, for simplicity, a single event models the interaction with all the Suppliers involved in a task session. For instance, the event `activate` models sending messages to all the suppliers to be activated for the task session. To activate a Supplier, the pair `activate`, followed by `ackActivate`, must be observed. If the Supplier needs to be attached, the pair `attach`, followed by `ackAttach`, will also be observed. Otherwise, attachment is skipped – in FSP this is modeled by the hidden event `noAttach`. Notice that deactivating a Supplier is accomplished by a single message exchange, `deactivate`. Notice also that the timeout event `noSupplierReply`, in case the Supplier fails to acknowledge an `activate` or `attach`, resets the protocol – Section 4.6 explains how this timeout is handled within the EM. During monitoring, a Supplier issues periodic QoS reports, (represented by `pQoSreport`, since FSP events cannot start with a capital letter) and may receive an arbitrary number of adjustments to its resource bounds, `adjust`. Note that the protocol allows `activate` events during the monitoring phase. There are two reasons for this. The first is that, as a consequence of later `setup` requests, the *same* Supplier may receive additional activations for other services. The second reason is a feature of the simplification explained above, where the communication with all Suppliers for the task session is modeled as a single process: again as a consequence of later setup requests, *other* Suppliers may need to be activated. Naturally, deactivation resets the protocol.

4.6 EM

The EM plays a central role in intermediating between the user's needs, for which Prism acts as a proxy, and the applications and devices in the environment. As such, the EM keeps models of both the capabilities of the environment, and of the user's needs, as transmitted by Prism. In addition to the FSP model of the EM's behavior, this section shows a Z model of the state kept by the EM as a result of the interactions with both Prism and the Suppliers. This state model is only as detailed as necessary to clarify the effects of such interactions.

Figure 12 shows the FSP specification for the behavior of the EM. That behavior is the parallel composition of the process for creating task models, `CreateTaskModel`, of an arbitrary number of processes to update as many task models, `UpdateTaskModel`, and of the same number of processes to manage the corresponding configuration of Suppliers in the environment, `ManageEnv`. The processes `CreateTaskModel` and `UpdateTaskModel` state the EM's view of the processes `CreateSess` and `TaskSession`, respectively, in the Prism-EM protocols (they are

restated here for completeness of the specification of EM's behavior). Notice that the `noEMreply` event is not seen by the EM since it corresponds to a timeout within Prism. Notice also that the observation of the event `missQoSreports` prompts the EM to issue a `taskLayout` with a reconfiguration suggestion to Prism (more on this below).

```
||EM = ( CreateTaskModel
      || forall [id:TId] t[id]:UpdateTaskModel
      || forall [id:TId] t[id]:ManageEnv )
      /  { resetEM / {t[TId]}.resetEM }.

CreateTaskModel  = ( newTask -> t[id:TId].createdTask -> CreateTaskModel).

UpdateTaskModel  = ( createdTask       -> CreatedTaskModel ),
CreatedTaskModel = ( {budget,setup}    -> IssueLayout
                   | missQoSReports    -> IssueLayout      // EM initiative
                   | resetEM           -> CreatedTaskModel
                   | disband           -> RemoveTaskModel ),
IssueLayout      = ( taskLayout        -> CreatedTaskModel
                   | resetEM           -> CreatedTaskModel ),
RemoveTaskModel  = ( taskGone          -> UpdateTaskModel
                   | resetEM           -> CreatedTaskModel ).

ManageEnv    = ( setup      -> ActivateSupp
               | pQoSreport -> deactivate -> ManageEnv
               | resetEM    -> ManageEnv ),
ActivateSupp = ( activate   -> ( ackActivate     -> AttachSupp
                               | noSupplierReply -> IssueLayout )
               | deactivate -> IssueLayout
               | resetEM    -> ManageEnv ),
AttachSupp   = ( attach     -> ( ackAttach       -> IssueLayout
                               | noSupplierReply -> IssueLayout )
               | skipAttach -> IssueLayout
               | resetEM    -> ManageEnv ),
IssueLayout  = ( taskLayout -> MonitorSupp
               | resetEM    -> ManageEnv ),
MonitorSupp  = ( adjust         -> MonitorSupp
               | pQoSreport     -> MonitorSupp
               | missQoSReports -> IssueLayout   // report to Prism
               | resetEM        -> MonitorSupp
               | setup          -> ActivateSupp
               | disband -> deactivate -> {taskGone,resetEM} -> ManageEnv )
             \ {skipAttach}.
```

Figure 12. FSP specification for behavior of the EM

The glue between the two protocols Prism-EM and EM-Supplier is specified by the `ManageEnv` process. As before, for simplicity, the messages exchanged with all the Suppliers supporting one task are modeled as single events. For instance, the `activate` event corresponds to sending `activate` messages to all such Suppliers. The implementation of the EM can use standard concurrency mechanisms, such as barriers, to wait for the reception of all the relevant acknowledge messages, and only then, from the protocol specification point of view, consider that it observed the corresponding `ackActivate` event. The activation and attachment of Suppliers is triggered upon receiving a `setup` request. In the case all the acknowledgements are received, that is, upon the successful activation and attachment of the requested Suppliers, the EM issues a `taskLayout` with the complete configuration. In the case some of the acknowledgements time-out, the event `nosupplierReply` is observed, and a `taskLayout` is still issued but this time with a possibly incomplete configuration. In this latter case, the EM may try to find and activate alternative, possibly less optimal Suppliers before returning the `taskLayout`. Notice that a `setup` may request some of the Suppliers in the current configuration to be deactivated (see `ActivateSupp` process). No acknowledgment is needed for deactivation before issuing the updated `taskLayout`.

After the first `setup` for a task session, the EM monitors the configuration according to the `MonitorSupp` process. Active Suppliers in a configuration periodically issue QoS reports to the EM. The EM uses these reports to evaluate the utility of the current set of suppliers against possible alternatives in the environment and may come up with an advantageous reconfiguration. Furthermore, when the EM notices that a particular Supplier fails to issue QoS reports, it will try to replace that (presumably) faulty Supplier: in the FSP model this is represented by the event `missQoSreports`, leading to the `IssueLayout` process. In either case, and according to the autonomy policies for swapping Suppliers (refer to Section 3.1), the EM may have to confirm with Prism that the reconfiguration is appropriate/opportune before carrying out. Nonetheless, the EM has full autonomy to `adjust` the resource bounds on the Suppliers.

After each (re)configuration of the environment, made in response to a `setup` request, the EM updates a persistent checkpoint of its models. In case the EM implementation fails, those persistent checkpoints enable restarting the EM without having to reconfigure the environment from scratch. That is, the Suppliers can continue to support the user's task, while Prism, upon detecting the EM's lack of response will restart the EM and reissue any pending `setup` request. The handling of the `resetEM` event in the `ManageEnv` process captures the fact that an environment reconfiguration is transactional in the following sense: if the EM fails anywhere between a `setup` and the corresponding `taskLayout`, no intermediate state is recovered. In such a case, any Suppliers that were activated by the incomplete reconfiguration will be detected and deactivated by the EM: the EM will react to QoS reports from Suppliers it does not recognise as being active by sending them a `deactivate` message.

$[Id, Description, UtilityValue, Supplier]$

EMTaskModel

$$serviceDesc: Id \pfun Description$$
$$knownServices: \mathbb{P}\ Id$$
$$suppPrefs: \mathbb{P}\ Supplier \fun UtilityValue$$

$$\mathrm{dom}\ serviceDesc = knownServices$$

EMEnvModel

$$knownSuppliers: \mathbb{P}\ Supplier$$
$$supplierMapping: Id \pfun Supplier$$
$$activeServices: \mathbb{P}\ Id$$

$$\mathrm{dom}\ supplierMapping = activeServices$$
$$\mathrm{ran}\ supplierMapping \subseteq knownSuppliers$$

EM

$$EMTaskModel$$
$$EMEnvModel$$
$$bestChoice: Description \times \mathbb{P}\ Supplier \fun Supplier$$

$$activeServices \subseteq knownServices$$

Figure 13. Z model of the state kept by the EM as a result of the Prism-EM communication

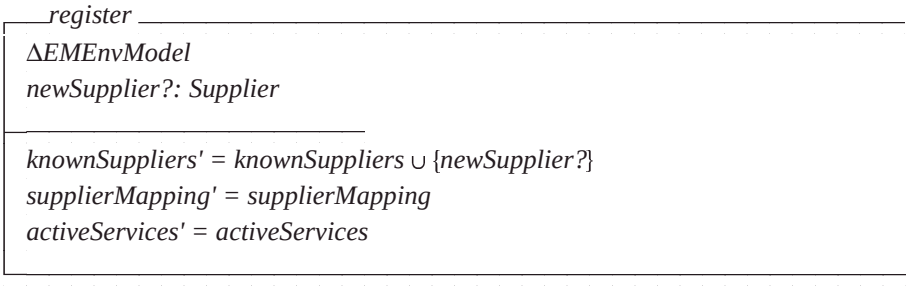

Figure 14. Z model of the effect of the `register` message on the state kept by the EM

Figure 13 shows the Z model of the state kept by the EM as a result of the interactions with Prism and the Suppliers. For each task session, the EM keeps both a model of the task as communicated by Prism, EMTaskModel, and of the environment that supports that task, EMEnvModel. The task model consists of two pieces: (1) a table of service descriptions indexed by service id, serviceDesc; and (2) the user preferences with respect to the choice of Suppliers for each service, suppPrefs. The model of the environment consists of two pieces: (1) the supplierMapping, which maps the id of each active service to the Supplier providing that service; and (2) the knowSuppliers set, which includes all the Suppliers that register with the EM, and is shared among all task sessions. For the sake of simplicity, the schema for the EM represents a single task model and environment model. Notice that the set of active services (the ones being currently provided by a Supplier) is a subset of the known services (the ones with a description transmitted by Prism). This is because Prism may explore a number of alternatives before settling on a set of services to support the user's task. Notice also that the bestChoice function corresponds to the algorithms within the EM that, given a service description, select the best fit among a given set of Suppliers. Figure 14shows the effect of a `register` message sent by a Supplier: only the set of known Suppliers in the EMEnvModel is updated with the new supplier.

Figure 15 shows the effect of a `budget` message sent by Prism. The purpose of this type of message is to run a "what if?" scenario against the current conditions in the environment. As such, a budget indicates the ids for the services to be hypothetically activated, deactivated (disbanded), or have the current Supplier replaced. Additionally, a `budget` piggybacks information for updating the task model: the relevant service descriptions, newServDescs, and an update on the user preferences with respect to Supplier choices, newSuppPrefs. Note that in the schema, the task model is affected, but the environment model is only observed. Consequently, the EM computes temporary values for the candidate services to be activated; the candidate Suppliers to choose from (all the known Suppliers, except for the ones that the user is unhappy with – the ones to be replaced); and the candidate configuration (the best choice of Suppliers for the candidate services). The utility value for the candidate configuration will be returned by the `taskLayout` message in reply to the `budget`.

budget

$\Delta EMTaskModel$
$\Xi EMEnvModel$
EM
$addServices?: \mathbb{P}\ Id$
$replaceServices?: \mathbb{P}\ Id$
$disbandServices?: \mathbb{P}\ Id$
$newServDescs?: Id \pfun Description$
$newSuppPrefs?: \mathbb{P}\ Supplier \pfun UtilityValue$
$utility!: UtilityValue$

$disj\ \{addServices?, replaceServices?, disbandServices?\}$
$addServices? \cup replaceServices? \cup disbandServices? \subseteq knownServices'$
$knownServices' = knownServices \cup \mathrm{dom}\ newServDescs?$
$serviceDesc' = serviceDesc \oplus newServDescs?$
$suppPrefs' = suppPrefs \oplus newSuppPrefs?$
$\textbf{let}\ candidateServices == activeServices \setminus disbandServices? \cup addServices?$
$\bullet\ \textbf{let}\ candidateSuppliers == knownSuppliers \setminus \{\ s: replaceServices? \bullet supplierMapping\ s\ \}$
$\bullet\ \textbf{let}\ candidateConfig == \{\ s: candidateServices \bullet bestChoice\ ((serviceDesc\ s), candidateSuppliers)\ \}$
$\bullet\ utility! = suppPrefs'\ candidateConfig$

Figure 15. Z model of the effect of the `budget` message on the state kept by the EM

setup

$\Delta EMTaskModel$
$\Delta EMEnvModel$
EM
$addServices?: \mathbb{P}\ Id$
$replaceServices?: \mathbb{P}\ Id$
$disbandServices?: \mathbb{P}\ Id$
$newServDescs?: Id \pfun Description$
$newSuppPrefs?: \mathbb{P}\ Supplier \pfun UtilityValue$

$disj\ \{addServices?, replaceServices?, disbandServices?\}$
$addServices? \cup replaceServices? \cup disbandServices? \subseteq knownServices'$
$knownServices' = knownServices \cup \mathrm{dom}\ newServDescs?$
$serviceDesc' = serviceDesc \oplus newServDescs?$
$suppPrefs' = suppPrefs \oplus newSuppPrefs?$
$activeServices' = activeServices \setminus disbandServices? \cup addServices?$
$knownSuppliers' = knownSuppliers$
$\textbf{let}\ candidateSuppliers == knownSuppliers \setminus \{\ s: replaceServices? \bullet supplierMapping\ s\ \}$
$\bullet\ supplierMapping' = supplierMapping$
$\setminus \{\ d: disbandServices? \cup replaceServices? \bullet (d \mapsto supplierMapping\ d)\ \}$
$\cup \{\ s: addServices? \cup replaceServices?$
$\bullet (s \mapsto bestChoice\ ((serviceDesc\ s), candidateSuppliers))\ \}$

Figure 16. Z model of the effect of the `setup` message on the state kept by the EM

Figure 16 shows the effect of a `setup` message sent by Prism. The purpose of this type of message is to set up or change the configuration of Suppliers currently supporting the user's task. Like a `budget`, a `setup` piggybacks information for updating the task model. However, a `setup` indicates the ids for the services to be effectively added or removed from the configuration. Consequently, the EM updates both the task and environment models (and of course, sends the appropriate messages to the affected Suppliers, as described in the protocol specification). The environment model is updated in the following way: (1) the set of active services is cleared of the disbanded service ids, and appended with the newly activated ones; (2) the supplier mapping is cleared of the mappings for the disbanded or replaced services, and added with the best choices for the services to be added, or to have their suppliers replaced, among the candidate Suppliers. As before, the candidate Suppliers are all the known Suppliers, except for the ones that the user is unhappy with.

5 Message contents

This Section shows example contents for each of the messages exchanged in the protocols specified in Section 4. Due to its tediousness, the specification of the format of all the exchanged messages is not covered here, but is available online as an XML Schema [9]. For the sake of example, a scenario for real-time speech-to-speech translation is used involving the configuration of three services. We will follow roughly the sequence of messages in Figure 5, Figure 7, and Figure 10. For clarity of explanation, some parts of each message are elided `<. . .>` and discussed separately. The complete example messages are also available online [9]. An explanation of the details dealing with service interconnection, namely `attach` messages, is postponed until Section 6; and the description of the materials (such as files) used by services, is the object of Section 5.1.

```
<register name="Janus" location="myURL" EMbind="myPortForEM" Pbind="myPortForPrism">

   <service type="speechRecognition" availableUnits="1" validFor="5" validUnit="minute">
      <...port description...>
      <warmup average="0.1" variance="0.1" unit="second"/>

      <QoSprofile>
         <QoSdimension name="latency" type="float"/>
         <QoSdimension name="vocabulary" type="enum"/>
      </QoSprofile>
      <resourceProfile>
         <resource name="cpu" type="integer"/>
         <resource name="bandwidth" type="integer"/>
      </resourceProfile>

      <QoSmap>
         <header>latency vocabulary cpu bandwidth</header>
         <units>second none   % Kbps</units>
         <point>0.05 small   30 250</point>
         <point>0.05 medium  50 250</point>
         <point>0.05 large   80 250</point>
         <point>0.1  small   20 200</point>
         <point>0.1  medium  40 200</point>
         <point>0.1  large   75 200</point>
         <point>0.2  small   20 180</point>
         <point>0.2  medium  30 180</point>
         <point>0.2  large   50 180</point>
         <point>0.5  medium  10 110</point>
         <point>0.5  large   30 110</point>
      </QoSmap>
   </service>
</register>
```

Figure 17. Example `register` message for a *speech recognition* service

Figure 17 shows an example `register` message issued by the Janus speech recognizer. The attributes in the main element are the `name`, to which the user refers in the supplier preferences; the network `location` of the Supplier (either an IP address or an URL); and two TCP ports where the Supplier is listening for messages coming in from the EM, `EMbind`, and from Prism, `Pbind`. A registration message contains one or more service announcements, which include the service `type`, and the number of available units. Typically, suppliers based on shareable software and devices will have several, or `unbound`, available units, while suppliers that involve non-shareable devices, such as sound input or output devices, have at most one unit available at any given time. The announcement validity attributes offer the EM a concrete criterion for the renovation of the registry of availability (see the *EMEnvModel* in Section 4.6). The body of a service announcement contains a description of the ports for service interconnection (see Section 6); an estimate for the warm-up time;[10] and the QoS characteristics of the Supplier. The latter include the QoS profile, which enumerates the QoS dimensions for the service; the resource profile, which enumerates the resources that the Supplier monitors and to which it adapts; and the QoS map, which maps a discrete set of typical fidelity points to the corresponding resource demands. During normal operation, the Supplier is responsible for benchmarking the component it wraps, and updating the information used to generate the QoS map – this is the same mechanism on which the periodic QoS reports are based (see Figure 26).

```
<newTask/>

<createdTask taskId="34"/>

<disband taskId="34"/>

<taskGone taskId="34"/>
```

Figure 18. Example session-framing messages for the speech-to-speech scenario

Figure 18 show example framing messages for a task session: `newTask` issued by Prism to create a new task; the corresponding EM reply, which includes the EM-generated `taskId`; down to the closing `disband`, and corresponding acknowledgment, `taskGone`.

Figure 19 shows example `budget` and `setup` messages. The allowed contents for these two messages are exactly the same – refer to Section 4.6 for a discussion of the interpretation and effects of each of these messages. The contents of these messages have three parts: first, a `change` element enumerates one or more operations to be performed on the elements of the task. These operations correspond to the *addServices*, *replaceServices* and *disbandServices* parameters in Figure 15 and Figure 16. Whereas for simplicity, Section 4.6 referred only to services, in general the elements requested to support a task are services, connections (see Section 6), and materials (see Section 5.1); each with a unique `id` within the task, and each described at some point in either a `budget` or `setup` message. The possible operations for the task elements are: `add` and `disband`, applicable to any task element, and `replace`, for services alone. Second, `budget` and `setup` messages piggyback zero or more descriptions for the relevant task elements: this corresponds to parameter *newServDescs* in Section 4.6. Third, both types of messages may carry an excerpt of supplier preferences – corresponding to the *newSuppPrefs* parameter. Notice that each message doesn't have to carry the complete supplier preferences because these, like the descriptions for task elements, are updated by functional overwriting.

[10] The average time that the Supplier needs to prepare for normal operation after receiving an `activate`. This corresponds to the term $W(p_s)$ – or $W(Janus)$, in this case – discussed in Section 3.2.

```
<budget taskId="34">
    <change>
        <service id="1" op="add"/>
        <service id="2" op="add"/>
        <service id="3" op="add"/>
    </change>

    <service id="1" type="speechRecognition" changeSupplier="ask">
        <...port description...>
        <...QoS preferences...>
    </service>

    <service id="2" type="languageTranslation" changeSupplier="goAhead">
        <...port description...>
        <...QoS preferences...>
    </service>

    <service id="3" type="speechSynthesis" changeSupplier="ask">
        <...port description...>
        <...QoS preferences...>
    </service>

    <...supplier preferences...>
</budget>

<setup taskId="34">
    <change>
        <service id="1" op="add"/>
        <service id="2" op="add"/>
    </change>
    <...service/material descriptions...>
    <...supplier preferences...>
</setup>
```

Figure 19. Example `budget` and `setup` messages for the speech-to-speech scenario

Service descriptions carry an `id`, which is local to the task, the `type` of service, and the attribute `changeSupplier` states whether the Environment Management has the autonomy to swap the supplying component as soon as a new one comes along with a higher utility (see Section 3.1). The `service` element also contains a description for the required ports, and the QoS preferences for the service (see below).

```
<utility combine="product">
    <QoSdimension name="latency" type="float">
        <function type="sigmoid" weight="1">
            <thresholds good="0.1" bad="2" unit="second"/>
        </function>
    </QoSdimension>
    <QoSdimension name="vocabulary" type="enum">
        <function type="table" weight="0.5">
            <entry x="large" f_x="1"/>
            <entry x="medium" f_x="0.95"/>
            <entry x="small" f_x="0.2"/>
        </function>
    </QoSdimension>
    <contextAtt name="distanceToUser" type="int">
        <function type="sigmoid" weight="1">
            <thresholds good="3" bad="10" unit="meter"/>
        </function>
    </contextAtt>
</utility>
```

Figure 20. Example QoS preferences for the *speech recognition* service

Figure 20 shows the QoS preferences for the *speech recognition* service in the example in Figure 19. Preferences are expressed for two QoS dimensions: the *latency* associated with the recogni-

tion of each utterance, and the size of the *vocabulary* that can be recognized (the more terms are searched, the more resource-demanding is the recognition). The function $F_{latency}$ is a sigmoid where the value over which the user is clearly unhappy with the response, `bad`, is 2 seconds, and the value under which the user does not perceive the delay as an issue, `good`, is 0.1 seconds (see Section 3.2). The function $F_{vocabulary}$ is a table that expresses that the user is happy with a large vocabulary, mostly happy with a medium vocabulary, and not so happy with a small vocabulary. The attributes `weight` in each of the two `function` elements correspond to $c_{latency}$ and $c_{vocabulary}$, respectively. A context attribute, *distance to user*, is shown to illustrate that preferences with respect to context attributes that pertain to the service can be expressed in the same way as QoS preferences (a separate report will cover Context Observation).

```
<utility combine="product">
    <supplier id="1">
        <function type="table" weight="1">
            <entry x="Janus" f_x="1"/>
            <entry x="Sphinx" f_x="1"/>
            <entry x="other" f_x="0.001"/>
            <entry x="change" f_x="0.1"/>
        </function>
    </supplier>
    <supplier id="2">
        <function type="table" weight="0.5">
            <entry x="BabelFish" f_x="1"/>
            <entry x="other" f_x="0.5"/>
            <entry x="change" f_x="1"/>
        </function>
    </supplier>
    <warmup>
        <function type="sigmoid" weight="1">
            <thresholds good="0.1" bad="2" unit="second"/>
        </function>
    </warmup>
</utility>
```

Figure 21. Example supplier preferences for the speech-to-speech scenario

Figure 21 shows the supplier preferences for the example in Figure 19, specifically for the services with id 1 and 2, as well as the warm-up term for the configuration. These elements correspond to the terms F_s, specifically $F_{speechRecognition}$ and $F_{languageTranslation}$, and F_w, respectively, in Section 3.2. The entry with `x="change"` under the function for each service *s* corresponds to the term h_s. In the example, the user will be happy if the language translator is exchanged, $h_{languageTranslation}$=1, but not if the speech recognizer is exchanged, $h_{speechRecognition}$=0.1. The user preferences with respect to the warm-up time are expressed as a sigmoid, where the `weight` attribute corresponds to c_w.

```
<taskLayout taskId="34" utility="0.85">

   <service id="1" op="added">
      <supplier name="Janus" location="SR-IPaddr" bind="SR-PortForPrism"/>
      <...estimated QoS...>
   </service>

   <service id="2" op="added">
      <supplier name="BabelFish" location="LT-IPaddr" bind="LT-PortForPrism"/>
      <...estimated QoS...>
   </service>
</taskLayout>
```

Figure 22. Example `taskLayout` message for the speech-to-speech scenario

Figure 22 shows an example `taskLayout` message corresponding to the `setup` in Figure 19. A `taskLayout` corresponding to a budget would also include an attribute indicating the `utility` of the environment for the requested task (shown italicized in the figure just for illustration purposes). In general, `taskLayout` messages contain one or more descriptions of the actions taken, or suggested, by the EM. Those actions are either in response to the operations requested under the `changes` element of the corresponding `budget` or `setup`, or proactively originated by the EM in response to changes in the environment. Each description uses the element name and `id` attribute to identify the component upon which the action was taken, or is suggested. The possible actions *taken* by the EM (in response to a `setup` or autonomously) are: `added`, `replaced`, and `disbanded`. The possible actions *suggested* by the EM (in response to a `budget` or autonomously) are: `add`, `replace`, and `disband`.[11] When adding or replacing services, the description includes a `supplier` element indicating its `name`, `location` and `bind` where it will be listening for Prism's messages. When adding or replacing either services or connections, the description also includes a QoS estimate (see below).

```
<utility>
   <QoSdimension name="latency" type="float">
      <numEstimate average="0.2" variance="0.1" unit="second"/>
   </QoSdimension>
   <QoSdimension name="vocabulary" type="enum">
      <enumEstimate value="medium" confidence="0.8"/>
   </QoSdimension>
   <contextAtt name="distanceToUser" type="int">
      <numEstimate average="4" variance="2" unit="meter"/>
   </contextAtt>
</utility>
```

Figure 23. Example estimated QoS for the *speech recognition* service

Figure 23 shows an example QoS estimate for the *speech recognition* service (`id="1"`) in the `taskLayout` message in Figure 22. For each dimension *d* in the QoS preferences for the service (see Figure 20), the EM includes an estimate of the QoS that the user can expect to observe – and which corresponds to the optimal fidelity points $\hat{f}_d$ found by the EM (see Section 3.2). An estimate for a numerical dimension includes the expected value, `average`, and `variance`. For an enumerated dimension, it includes the expected `value`, and the expected frequency of observations of that value, `confidence`. Note that estimates for context attributes can be encoded in the same fashion.

Figure 24 shows examples of the messages used to activate, attach, adjust and deactivate the suppliers, as well as the corresponding acknowledgements. All messages exchanged back and forth with Suppliers include the `taskId`, as well as the `location` and `bind` where the Supplier is listening for messages. An `activate` message includes one or more `service` elements corresponding to the services being marshaled from the Supplier. The attributes of the `service` are the `id` within the task, for ease of later reference, as well as the service `type`. The `service` elements inside both the `activate` and `adjust` messages contain resource constraints (see below). A Supplier acknowledges activation as soon as it successfully acquires the resources necessary to supply the requested service. That is, the Supplier doesn't have to wait to complete the warm-up in order to send an `ackActivate`, which echoes the `id`s of the successfully marshaled services. The `attach` and `ackAttach` messages are covered in Section 6. Finally, the

[11] Strictly, the EM has to include a description for each action, but a conservative EM may also include descriptions for the all other components (services, materials, and connections) supporting the task, under the `op` value `noOp`.

`deactivate` message lists the services (identified by `id`) to be deactivated, among those currently marshaled from the target Supplier for task `taskId`.

```
<activate taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <service id="1" type="speechRecognition">
      <...resource constraints...>
   </service>
</activate>

<ackActivate taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <service id="1"/>
</ackActivate>

<attach taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <...connection description...>
</attach>

<ackAttach taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <...connection confirmation...>
</ackAttach>

<adjust taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <service id="1">
      <...resource constraints...>
   </service>
</adjust>

<deactivate taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <service id="1"/>
</deactivate>
```

Figure 24. Example supplier-configuration messages for the speech-to-speech scenario

```
<constraints>
   <resource name="cpu" type="integer">
      <numEstimate average="30" variance="10" unit="%"/>
   </resource>
   <resource name="bandwidth" type="integer">
      <numEstimate average="180" variance="40" unit="Kbps"/>
   </resource>
</constraints>
```

Figure 25. Example resource constraints for the *speech recognition* service

Figure 25 shows the resource constraints included in the `activate` for the *speech recognition* service in Figure 24. These constraints are obtained by the EM when calculating the optimal fidelity points (see Figure 23), taking into account the relationship between the two, as expressed in the service registration (see the QoS map in Figure 17). A constraint is issued for each resource in the resource profile of the service, with the same form as the estimates for the QoS: typically, numeric estimates with an `average`, a `variance`, and a `unit`; or an enumerated estimate with a `value`, and a `confidence`.

Figure 26 shows an example QoS report issued by the Supplier for the speech recognition service after the activation shown in Figure 24. Like other messages exchanged with the Suppliers, `QoSreport` indicates the corresponding `taskId`, as well as the `location` and `bind` where the Supplier is listening for messages. A `QoSreport` includes a report for one or more services (as many as being supplied by the Supplier for the task at hand) identified by the service `id`. Each report has the same format as the QoS estimate in Figure 23, but its contents reflect the actual values of the QoS being offered to the user, as monitored by the Supplier.

```
<QoSreport taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <service id="1">
      <utility>
         <QoSdimension name="latency" type="float">
            <numEstimate average="0.3" variance="0.1" unit="second"/>
         </QoSdimension>
         <QoSdimension name="vocabulary" type="enum">
            <enumEstimate value="medium" confidence="0.9"/>
         </QoSdimension>
         <contextAtt name="distanceToUser" type="int">
            <numEstimate average="3" variance="1" unit="meter"/>
         </contextAtt>
      </utility>
   </service>
</QoSreport>
```

Figure 26. Example `QoSreport` message for the *speech recognition* service

```
<setState taskID="34" location="SR-IPaddr" bind="SR-PortForPrism">
   <service id="1" type="speechRecognition">
      <settings>
         <speechRecognition language="English"/>
      </settings>
      <...QoS preferences...>
   </service>
</setState>

<getState taskId="34" location="SR-IPaddr" bind="SR-PortForPrism">
   <service id="1" type="speechRecognition"/>
</getState>

<stateSnapshot taskId="34" location="SR-IPaddr" bind="SR-PortForPrism">
  <service id="1" type="speechRecognition">
    <settings>
      <speechRecognition language="English"/>
    </settings>
  </service>
</stateSnapshot>
```

Figure 27. Example set/get state messages for the *speech recognition* service

Figure 27 shows examples for the messages exchanged between Prism and the Suppliers when setting and getting the user-level state (see Section 2.1) for the services within the task budgeted in Figure 19. Like other messages exchanged with the Suppliers, these indicate the corresponding `taskId`, as well as the `location` and `bind` where the Supplier is listening for messages. The `setState` message is sent to the Supplier for the *speech recognition* service, after Prism receives the `taskLayout` in Figure 22. A `setState` includes the state for one or more services (as many as being supplied by the Supplier for the task at hand), identified by the service `id`. The service `type` may be included, as shown in the example, although this is redundant information, since the `activate` message already made the Supplier aware of the correspondence between the service `id` and the `type`. The service-specific settings are included under the `settings` element, and if the service uses some material, its description will be under a `material` element (see Section 5.1). A `setState` also includes the QoS preferences for the service. A `getState` message enumerates the services for which Prism wishes to obtain a `stateSnapshot`, each under a `service` element. Again, service `id` is used for identification, and service `type` may optionally be included. A `stateSnapshot` message includes the up-to-date values of the settings for the requested services, and state of the materials, as modified during the operation of the service by the user.

5.1 Materials & Data Staging

The infrastructure can enact data staging by transferring remote files in advance to the environment where the user will be carrying out his task. For that, materials are added to the task elements requested by Prism to the EM. Similarly to services, adding a material to a `budget` request produces an estimate: in this case, an estimate for the transfer time. Adding a material to a `setup` request starts the process of transferring the remote file to the local environment. In case the file is being accessed for read/write, as opposed to read only, the EM will transfer the file back upon a request to `disband` the material (or to disband the task as a whole). The reference to materials (files, data streams, databases…) accessed remotely is passed to the relevant Suppliers alone (not to the EM) within a `setState` message.

```
<budget taskId="36">
   <change>
      <service id="1" op="add"/>
      <service id="2" op="add"/>
      <material id="3" op="add"/>
   </change>

   <service id="1" type="videoPlaying" changeSupplier="ask">
      <...QoS preferences...>
   </service>

   <service id="2" type="textEditing" changeSupplier="ask">
      <...QoS preferences...>
   </service>

   <material id="3">
      <file location="rootURL" bind="directory" name="file" rw="rw" protocol="http"/>
      <usedBy serviceId="2"/>
   </material>

   <...supplier preferences...>
</budget>
```

Figure 28. Example `budget` *message* with materials

Figure 28 shows an example `budget` request for the task of reviewing video clips. Two services are budgeted, *text editing* and *video playing*, along with the text document to be edited. `Material` elements have a unique `id` within the space of task elements, and all materials requested to the EM are files. There are one or more `file` elements within the `material`, corresponding to alternative ways to access the file. A `file` element indicates the remote `location`, `bind`, and `name` of the file, as well as the `protocol` to access it (http, ftp, or file sharing services, such as CVS). For example, for ftp, `location` indicates the network server, `bind` the directory within the server, and `name` the name of the file. The attribute `rw` indicates whether the file will be accessed for read only, `r`, or for read/write, `rw`. In the latter case, the EM makes sure the file is transferred back after being disbanded. The EM takes the `usedBy` information as a hint to place the local copy of the file in a storage device easily accessible by the Supplier for the indicated service. There may be as many `usedBy` elements as many services share that same file – in which case the EM will place the material in commonly accessible storage.

Figure 29 shows an example `taskLayout` message corresponding to the `budget` in Figure 28. The description for the requested material includes the `location` (effective only after a `setup`) in the local environment as well as the access `protocol`, typically the file system, `fs`. Additionally, the `material` description includes an estimate for the delay in transferring the file, starting from the moment when the `setup` is received: the `availability` element follows the same format as the `numEstimate` element in QoS estimates.

```
<taskLayout taskId="36" utility="0.95">

   <service id="1" op="add">...</service>
   <service id="2" op="add">...</service>

   <material id="3" op="add">
      <file location="localDirectory" name = protocol="fs"/>
      <availability average="10" variance="5" unit="second"/>
   </material>
</taskLayout>
```

Figure 29. Example `taskLayout` message with materials

```
<setState taskID="36" location="TE-IPaddr" bind="TE-PortForPrism">
   <service id="1" type="textEditing">
      <settings>
         <pane height="260" width="200" unit="mm"/>
         <spelling enabled="true" ignoreAllCaps="true"/>
         <editing overstrike="false" replaceSelection="true"/>
      </settings>
      <...QoS preferences...>
      <material id="3">
         <file location="localDirectory" protocol="fs"/>
         <state>
            <text scroll="0" zoom="100" cursor="15"/>
         </state>
      </material>
   </service>
</setState>
```

Figure 30. Example `setState` message for *text editing* a local file

```
<setState taskID="36" location="VP-IPaddr" bind="VP-PortForPrism">
   <service id="2" type="videoPlaying">
      <settings>
         <playingOptions multiplePanes="yes" autoresize="yes"/>
      </settings>
      <...QoS preferences...>
      <material id="4">
         <file location="URL" bind="directory" name="video" format="mp2" protocol="http"/>
         <file location="URL" bind="directory" name="video" format="mp4" protocol="http"/>
         <state>
            <video playing="pause" cursor="00:01:15" zoom="100"/>
            <audio level="40" mute="no"/>
         </state>
      </material>
      <material id="5">
         <stream location="URL" bind="directory" name="video" format="avi" protocol="http"/>
         <state>
            <video playing="play" cursor="" zoom="200"/>
            <audio level="60" mute="no"/>
         </state>
      </material>
   </service>
</setState>
```

Figure 31. Example `setState` message for *video playing* a remote file and a remote stream

Figure 30 shows an example `setState` message for the *text editing* service, initially budgeted in Figure 28. Note that the `service` element now includes a description for each of the materials being processed by the service. A material description is comprised of, first, the information on how to access the material – in the example, the `file` element returned by the EM in the `taskLayout` in Figure 29 – and second, the `state` of the material. Note that the `settings` of

the service include three groups of settings, applicable to all the materials, and that the state of the material is specific to the file being edited (see Section 2.1).

Figure 31 shows an example `setState` message for the *video playing* service, initially budgeted in Figure 28. For the sake of example, two materials are being used by this service, both accessed remotely – and therefore, not requested to, and not prefetched by the EM. The first is a remote video file, and the second is a video stream. Note that the location information follows the exact same format used by Prism to request data staging to the EM. However, for remote access, multiple formats can be used to exploit tradeoffs between video quality and bandwidth utilization. The `format` attributes of `file`/`stream`, enumerate the available data formats.

6 Interconnecting services

Service interconnection is reflected at the three levels of the infrastructure shown in Figure 2. At the Task Management layer, the user describes *which* services he wants interconnected – and those requirements are propagated down in the `budget` requests issued by Prism. At the Environment Management Level, the EM interprets the required interconnections, and upon a `setup`, issues `attach` messages to the Suppliers of the services to be interconnected (see Section 4.5). Each Supplier uses the communications infrastructure in the environment (networks and middleware) according to the specific characteristics of the encapsulated component. The Suppliers at the endpoints of a connection are responsible for gathering monitoring information (see [3]) and report the QoS of the connection back to the EM, piggybacked on their periodic QoS reports.

Descriptions built at the Task Management level reflect the depth of understanding of the user about services and service interconnection, rather than the depth of understanding of a software architect or system's specialist. For example, in the physical world, to hook up a regular PC the user needs to connect the power outlet to the PC's power input, using the power cable, the video output to the monitor's video in, using the video cable, and so on. The user doesn't need to be familiar with the pin layout of the video cable, or with the specifications of the electrical signals that go in each pin. In Software Architecture, specifications of connectors can be fairly complex, including the port (interface) signatures, interaction protocols, and so on. In the Aura infrastructure, this depth of knowledge resides at the Environment Management level: the EM accounts for interconnection compatibility, and the Suppliers (and components they wrap) account for the specifics of the interfaces and protocols.

Consequently, the concepts to be captured when describing service interconnection at the Task Management level are distinct from those captured in Architectural Description Languages such as Acme, or their XML-based counterparts, such as xArch [4,5]. The main concept is that of the dynamic establishment of (point-to-point) *connections* between services. This concept can be used while being oblivious of *how* that connection will be realized: opening a session on a point-to-point *connector*, activating a publish-subscribe mechanism over an event bus (multi-point connector), etc. Under the premise of leaving the connector-specific knowledge at the Environment Management level, Prism requests connections by identifying the services to be connected and the port *types* on those services. It is up to each Supplier to decide whether the connection requires the use of a dedicated port, or if it can be multiplexed as a new session on a shared port.

```
<budget taskId="34">
   <change>
      <service id="1" op="add"/>
      <service id="2" op="add"/>
      <service id="3" op="add"/>
      <connection id="4" op ="add"/>
      <connection id="5" op ="add"/>
   </change>

   <service id="1" type="speechRecognition" changeSupplier="ask">
      <port type="textOut"/>
      <...QoS preferences...>
   </service>

   <service id="2" type="languageTranslation" changeSupplier="goAhead">
      <port type="textIn"/>
      <port type="textOut"/>
      <...QoS preferences...>
   </service>

   <service id="3" type="speechSynthesis" changeSupplier="ask">
      <port type="textIn"/>
      <...QoS preferences...>
   </service>

   <connection id="4" type="pipe">
      <attach>
         <from serviceId="1" port="textOut"/>
         <to serviceId="2" port="textIn"/>
      </attach>
      <...QoS preferences...>
   </connection>

   <connection id="5" type="pipe">
      <attach>
         <from serviceId="2" port="textOut"/>
         <to serviceId="3" port="textIn"/>
      </attach>
      <...QoS preferences...>
   </connection>

   <...supplier preferences...>
</budget>
```

Figure 32. Example of a `budget` with interconnections

Figure 32 shows an example `budget` request with three interconnected services: *speech recognition*, (textual) *language translation*, and *speech synthesis*. To achieve real-time translation of speech into speech, the speech recognition output is piped to the input of the language translation, whose output is in turn piped into the input of speech synthesis. The notion of *connection* is added to the `budget` request, with a format and treatment similar to the notion of service. Like services, connections have an associated type, such as *pipe*, or *RPC*; and QoS preferences, with QoS dimensions such as bandwidth and latency.[12] The `changes` element refers to the connections to be added or removed in the same way that it refers to the services to be added or removed. The EM uses the type information in port descriptions (see below) and connection requests to decide on marshaling and interconnection compatibility. The QoS preferences for the connections are factored into the framework in Section 3.2 by adding the connections to the set *a*, with the terms corresponding to the supplier preferences set to one.

[12] As with service types, distinct connector types typically have a distinct set of QoS dimensions.

```
<port type="textIn" TCPbind="LT-textInPort" DLLbind="LT-textInDll"/>
<port type="textOut"/>
```

Figure 33. Example port description in the `register` message for a *languageTranslation* service

Figure 33 shows an example port description made within the `register` message issued by a Supplier of a *languageTranslation* service (see Figure 17). There is one `port` element for each port `type` that the Supplier supports. For ports that can accept messages/data, there is one bind attribute for each different way the port is reachable – networking, middleware such as RMI, EJB, etc. In the example, `textOut` ports are strictly originators of information, while the `textIn` ports are accessible either by a TCP connection (the value of `TCPbind` is the TCP port where the Supplier will be listening for messages), or by a DLL that can be loaded by the connected Supplier, if running on the same device, which supports the `textIn`-specific methods.[13] Depending on supplier locations and available middleware, the EM decides on which infrastructure to use for establishing the connections.

```
<attach taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <service id="1" type="speechRecognition">
      <connect id="5" port="textOut" connector="pipeOverTCP.dll"
               location="LT-IPaddr" TCPbind="LT-textInPort"/>
   </service>
</attach>

<ackAttach taskId="34" location="SR-IPaddr" bind="SR-PortForEM">
   <service id="1" type="speechRecognition">
      <connect id="5"/>
   </service>
</ackAttach>

<attach taskId="34" location="LT-IPaddr" bind="LT-PortForEM">
   <service id="2" type="languageTranslation">
      <listen id="5" port="textIn" TCPbind="LT-textInPort"/>
      <connect id="6" port="textOut" connector="pipeOverTCP.dll"
               location="SS-IPaddr" TCPbind="SS-textInPort"/>
   </service>
</attach>

<ackAttach taskId="34" location="LT-IPaddr" bind="LT-PortForEM">
   <service id="2" type="languageTranslation">
      <listen id="5"/>
      <connect id="6"/>
   </service>
</ackAttach>

<attach taskId="34" location="SS-IPaddr" bind="SS-PortForEM">
   <service id="3" type="speechSynthesis">
      <listen id="6" port="textIn" TCPbind="SS-textInPort"/>
   </service>
</attach>

<ackAttach taskId="34" location="SS-IPaddr" bind="SS-PortForEM">
   <service id="3" type="speechSynthesis">
      <listen id="6"/>
   </service>
</ackAttach>
```

Figure 34. Example connection messages for the speech-to-speech scenario

[13] Some middleware infrastructures use DLLs as proxies for remote components, effectively hiding the connection from the "client" components. From the perspective of managing Ubicomp environments, exposing the connection – and its properties – seems preferable.

Figure 34 shows the `attach` messages for connecting the three Suppliers in the speech-to-speech scenario, paired with their confirmation replies to the EM, `ackAttach`. Each `attach` message contains the connection request for one or more services (as many as being supplied by the Supplier for the task at hand). Each `service` element contains as many `connect` elements as the port types in that service that sit on the originating end of the connection; and as many `listen` elements as the port types in the service that sit on the receiving end of the connection. In the example that follows from Figure 32, connection 5 generates a `connect` element for *speech recognition* Supplier and a `listen` element for the *languageTranslation* Supplier. Notice that the `attach` message directed to the *languageTranslation* Supplier contains both the `listen` for the `textIn` port (connection 5) and the `connect` for the `textOut` port (connection 6). In this example, the connection is asymmetric: one port is a listener, while the other is an originator of information. In symmetric situations, such as peer-to-peer communication, the EM initiates the connection starting at an arbitrary end – for hybrid ports the Suppliers must be ready to either initiate or accept a connection request.

The attributes of a `connect` element are the `id` of the connection, the originating `port` type, a DLL that the Supplier may load if it lacks the means to access the `connector` infrastructure directly; and the `location` and bind for that identify the receiving end to that connector infrastructure. In the example, the connector infrastructure is TCP, and that is reflected in the name of the DLL to access it, in the contents of the `location` (an IP address), in the name of the bind attribute, `TCPbind`, and in its contents (the listener's TCP port). The attributes of a `listen` element are the `id` of the connection, the accepting `port` type, and the bind that identifies both connector infrastructure and the receiving end on that infrastructure. In the example the choice of the connector infrastructure is reflected in the name of the bind attribute, `TCPbind`, and in its contents (the TCP port to listen at). The `ackAttach` messages confirm the successful establishment of each end of a connection by echoing the `connect` and `listen` elements in the request, with the associated connection `id`.

```
<taskLayout taskId="34" utility="0.75">

   <service id="1" op="add">
      <supplier name="Janus" location="SR-IPaddr" bind="SR-PortForPrism"/>
      <...estimated QoS...>
   </service>
   <service id="2" op="add">...</service>
   <service id="3" op="add">...</service>

   <connection id="4" op="add">
      <...estimated QoS...>
   </connection>
   <connection id="5" op="add">...</ connection>
</taskLayout>
```

Figure 35. Example `taskLayout` message for the speech-to-speech scenario with interconnections

Figure 35 shows the `taskLayout` corresponding to the `budget` in Figure 32. Note that the connection elements appear with attributes similar to the services, and include an estimate for the QoS offered by the connection.

7 Service decomposition

There are several pertinent questions concerning service decomposition: foremost, what is the appropriate level for the vocabulary of services to be requested from the environment? For instance, in the speech-to-speech translation example in Sections 5 and 6, should Prism ask for a

configuration of three services, as illustrated, or for a single *speech-to-speech* service? In the latter case, should we rely on all the environments on the user's path to have a Supplier for such service?[14] If not, where in the infrastructure resides the knowledge to assemble the requested service out of the parts that exist in a particular environment? Once the composite service is assembled, should the internal structure be exposed? If yes, to what extent should it be exposed, and to which components/levels of the infrastructure?

Concerning the level of the vocabulary of services, it is not likely that a single definitive answer will ever crystallize. On the demand side, users with different degrees of expertise will ask for things at different levels: an inexperienced used will try to get more abstract services, hiding internal details as much as possible, while an expert user may want to have more control over which, and how the parts are configured. The *same* user may want to have more control over the structure supporting a critical task, but be willing to take an off the shelf solution when that same kind of task is low priority. On the supply side, it is to be expected that sophisticated environments, such as smart rooms, will have higher-level, well-tuned components, while poorer environments, such as handhelds, will have a collection of generic parts that can be assembled to deliver a similar function but in a less polished way.

The knowledge about assembling services out of parts resides in the Environment Management layer. This is a natural consequence of the distribution of responsibilities in Figure 2. The Task Management layer knows as much, or as little, about *what* the user wants as the user tells it: if the user asks for a *speech-to-speech* service, that's what Prism will try to obtain from the environment; if the user asks for three services interconnected in a certain way, that also is what Prism will try to obtain for the user. *How* a requested service can be assembled in a particular environment depends on the capabilities of that environment. In order to provide a structured way of adding knowledge about service decomposition to the Environment Management layer, that knowledge is captured in Suppliers (of composite services). Since all the capabilities of the environment, including the ones pertaining to service composition, are captured in Suppliers, the (design, if not the implementation of the) EM can be reused across distinct environments.

The structural information concerning a composite service is exposed all the way to the Task Management layer. Whenever a single requested service is actually being provided by an assembly of Suppliers, the user may have to interact with several applications, meaning several UIs, rather than an integrated one. Prism has to be aware of *what* exactly is being provided in lieu of what was requested, even if for nothing more than explaining it to the user.

The remainder of this section describes how the Aura infrastructure supports service decomposition. As before, we'll use the speech-to-speech translation example for illustration purposes. Figure 36 shows an event sequence diagram that illustrates the typical lifecycle for a Supplier of a composite service (for simplicity, a *composite Supplier*). Composite Suppliers register their services like any other Supplier, except that the map between QoS and resource demand is not included. Given that the QoS map of the composite Supplier is the cross product of the QoS maps for each of the parts, the reason for not including it in the registration is twofold: first, the cross product can be quite voluminous. Second, and most importantly, an accurate QoS map is only determined after a Supplier is chosen for each of the services in the composition. Instead of showing a QoS map, the registration of a composite Supplier shows its internal structure in a format similar to a request for services in a `budget` or `setup` (see Figure 38).

[14] These questions are, of course, recurrent, no matter which level we choose for the vocabulary. For instance, will an integrated *speech recognition* service be available in all environments, or should the infrastructure be prepared to compose it out of lower level-parts, say, *sound input* and *speech processing*?

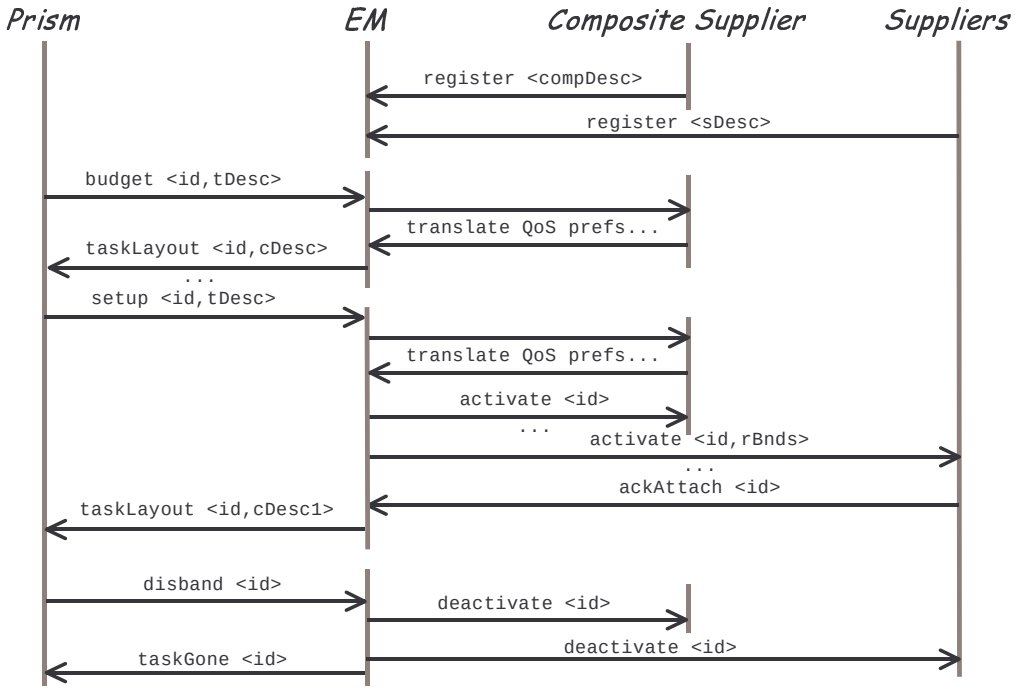

Figure 36. Event sequence diagram for the life-cycle of a composite Supplier

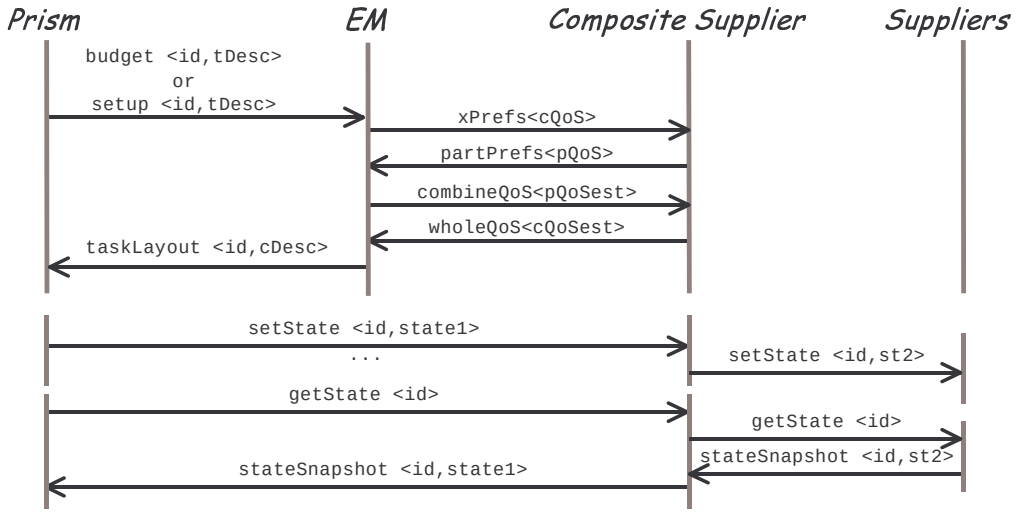

Figure 37. Event sequence diagram for the role of a composite Supplier

When a composite service is requested by Prism, the EM expands the request by in-lining the internal structure of the composite service(s), and then it searches for all the appropriate Suppliers. In order to determine the optimal fidelity points for the parts, the EM must first translate the QoS preferences expressed for the composite service into QoS preferences for each of the parts. Furthermore, once the optimal fidelity points are determined for the parts, they should be combined to provide the composite QoS estimate to Prism. The knowledge to perform these translations is specific to the composite service, and as such, it is held by the composite Supplier – the interactions for the EM to obtain such translations are discussed below. Once the optimal configuration is determined, the EM activates all the Suppliers, including the composite Supplier and all its

parts, and interconnects the parts according to the structure sent in the `register` for the composite Supplier. Note that the composite Supplier will typically receive no resource bounds, since it's role is to assist the EM and Prism in configuring the parts, rather than to directly support the user's task. When disbanding the task, the EM deactivates all Suppliers, including the composite Supplier and all its parts.

The role of the composite Supplier is to assist the EM and Prism in configuring the parts of the composite service – see Figure 37. This role includes (a) translating the QoS preferences for the composite service into QoS preferences for each of the parts – messages `xPrefs` and the reply `partPrefs`; (b) combining the QoS estimates of the parts into a QoS estimate for the composite service – messages `combineQoS` and the reply `wholeQoS`; (c) decomposing the user-level state of the composite service into the states for each of the parts – propagation of `setState` messages to the parts; and (d) combining the user-level state of each of the parts into the state of the composite service – propagation of `getState` messages to the parts, followed by the aggregation of the `stateSnapshot` replies.

```
<register name="AcmeSS" location="Acm-IPaddr" EMbind="Acm-PortForEM" Pbind="Acm-PortForPrism">

   <service type="speechToSpeech" availableUnits="1" validFor="5" validUnit="minute">
      <warmup average="3" variance="0.5" unit="second"/>

      <decomposition>
         <service id="1" type="speechRecognition" changeSupplier="ask">
            <port type="textOut"/>
         </service>

         <service id="2" type="languageTranslation" changeSupplier="goAhead">
            <port type="textIn"/>
            <port type="textOut"/>
         </service>

         <service id="3" type="speechSynthesis" changeSupplier="ask">
            <port type="textIn"/>
         </service>

         <connection id="4" type="pipe">
            <attach>
               <from serviceId="1" port="textOut"/>
               <to serviceId="2" port="textIn"/>
            </attach>
         </connection>

         <connection id="5" type="pipe">
            <attach>
               <from serviceId="2" port="textOut"/>
               <to serviceId="3" port="textIn"/>
            </attach>
         </connection>

         <...supplier preferences...>
      </decomposition>
   </service>
</register>
```

Figure 38. Example `register` message for a *speech-to-speech* service

Figure 38 shows an example `register` message issued by a composite Supplier offering the *speech-to-speech* translation service. As discussed above, a QoS map for the composite service is not included, and the `decomposition` of the service is shown instead. Specifically, *speech-to-speech* is supported by the assembly of the *speech recognition*, *language translation*, and *speech synthesis* services, already discussed in Figure 32. However, unlike a service request in the `budget` message, a `decomposition` element does not include the QoS preferences for the parts:

those will be determined once the QoS preferences for the composite service are known. Nevertheless, the composite Supplier may express *supplier preferences* for its parts, presumably based on empirical knowledge of which Suppliers work well together to provide the composite service. Note that the supplier preferences for the parts of a composite service are not included in a `budget` or `setup` request, since the Task Management is deliberately oblivious of how the composite service will be assembled. Furthermore, the composite Supplier expresses whether the Environment Management has the autonomy to swap the Supplier for a part as soon as a new one comes along with a higher utility (see Section 3.1). Specifically, if the `changeSupplier` attribute for the composite service is `ask`, then the EM applies the swapping policies expressed in the `decomposition`; but a `goAhead` value for the composite service overrides the swapping policy for all parts to `goAhead`.

```
<budget taskId="34">
    <change>
        <service id="1" op="add"/>
    </change>

    <service id="1" type="speechToSpeech" changeSupplier="ask">
        <...port description...>
        <...QoS preferences...>
    </service>
    <...supplier preferences...>
</budget>
```

Figure 39. Example `budget` message for the composite speech-to-speech scenario

Figure 39 shows an example budget for the (composite) *speech-to-speech* service. Notice that at this stage Prism is oblivious to whether a requested service will be provided by a single Supplier, or whether it will be treated as composite by the current environment. (Compare the detail of this `budget` with the one in Figure 32.) Notice that the QoS preferences under the `service` description correspond to the service (as a whole) – more on this below.

```
<taskLayout taskId="34" utility="0.8">
   <service id="1" op="add">
      <supplier name="AcmeSS" location="Acm-IPaddr" bind="Acm-PortForPrism"/>
      <...estimated QoS...>

      <decomposition>
         <service id="1.1" op="add">
            <supplier name="Janus" location="SR-IPaddr" bind="SR-PortForPrism"/>
            <...estimated QoS...>
         </service>
         <service id="1.2" op="add">...</service>
         <service id="1.3" op="add">...</service>

         <connection id="1.4" op="add">
            <...estimated QoS...>
         </connection>
         <connection id="1.5" op="add">...</connection>
      </decomposition>
   </service>
</taskLayout>
```

Figure 40. Example `taskLayout` message for the composite speech-to-speech scenario

Figure 40 shows the `taskLayout` corresponding to the budget in Figure 39. Notice that in correspondence to the request a singe `service` is listed, along with the information for accessing the Supplier for that service and the estimated QoS. However, the presence of a `decomposition` element indicates that the service is being provided by an assembly of Suppliers, rather than by a

single Supplier. Compare the detail under the `decomposition` element with the `taskLayout` in Figure 35. The services being provided within the decomposition are identified after the `id`s in the corresponding `register` for the composite service, prefixed by the service `id` in the `budget` request. Note that this structure is merely informative to Prism, since setting and getting the user-level state is done through the Supplier for the requested service, whether or not it is a composite Supplier. Notice also that this decomposition scheme is recursive: if, for instance, service `1.2` were itself being provided by a composite Supplier, it would contain a `decomposition` element containing services `1.2.1`, etc. Of course, Prism can drill down such a structure as much, or as little, as the user's curiosity demands.

```
<register name="Asupplier" <...access information...>>

   <service type="A" availableUnits="1" validFor="5" validUnit="minute">
      <...warmup.../>

      <decomposition>
         <service id="1" type="B" changeSupplier="goAhead">
            <port type="textIn"/>
            <port type="textOut"/>
         </service>

         <service id="2" type="C" changeSupplier="goAhead">
            <port type="textIn"/>
            <port type="textOut"/>
         </service>

         <connection id="3" type="pipe">
            <attach>
               <from serviceId="1" port="textOut"/>
               <to serviceId="2" port="textIn"/>
            </attach>
         </connection>

         <exposePorts>
            <service id="1" port="textIn"/>
            <service id="2" port="textOut"/>
         </exposePorts>

         <...supplier preferences...>
      </decomposition>
   </service>
</register>
```

Figure 41. Example `register` message for an interconnectable composite service

Although not illustrated in the speech-to-speech example, composite services may be interconnected, like any other services. For that, the registration of a composite service indicates which ports in the parts are exposed as ports of the composite service. Figure 41 shows an example registration for a service `A`, composed of services `B` and `C`. Internally, `B` and `C` form a pipeline, with the output of `B` connected to the input of `C`. Additionally, the input port of `B` and the output port of `C` are exposed (as ports of `A`), which can then be used for interconnection, as if they were declared in a `<…port description…>` section for `A` (see Figure 33 – and note that the binding information for the exposed ports is derived from the port description in the registration issued by the Suppliers for the parts).

Figure 42 shows example messages exchanged between the EM and a composite Supplier for the translation of QoS preferences. In the example shown, the QoS preferences for the *speech-to-speech* service extracted from the `budget` in Figure 39 are included in the message `xPrefs`. The reply `partPrefs` contains the QoS preferences for each of the parts `1` – `5` of the composite service registered in Figure 38. To perform this translation, the composite Supplier needs to know

(a) the mapping between each dimension in the QoS preferences for the composite service and the QoS dimensions of the parts; and (b) the relationship between the QoS dimensions of the parts. Suppose, for instance, that the QoS preferences for *speech-to-speech* include two dimensions: the overall latency of the translation (of each utterance) and the accuracy of such translation. Suppose further that the overall latency is the sum of the individual latencies, and that there is a known proportion between the latencies of each part. Given the `good` and `bad` limits of the total latency, the `good` and `bad` limits for the latency of each part are easily calculated. Similarly, suppose that the overall accuracy is given by a weighted sum (or by the product) of individual accuracies. Given a known relationship between individual accuracies, the preferences are easily translated. Notice how specific this kind of knowledge is to each composite service and to the assembly of the parts supporting it. Note also, that this translation is requested as needed. That is, if consecutive `budget` or `setup` requests refer to the same QoS preferences (for the composite service) there is no need to translate those every time.

```
<xPrefs location="Acm-IPaddr" bind="Acm-PortForEM"">
   <service type="speechToSpeech">
      <...speech-to-speech QoS preferences...>
   </service>
</xPrefs>

<partPrefs location="Acm-IPaddr" bind="Acm-PortForEM"">
   <service type="speechToSpeech">
      <service id="1">
         <...speech-recognition QoS preferences...>
      </service>
      <service id="2">
         <...language-translation QoS preferences...>
      </service>
      <service id="3">
         <...speech-synthesis QoS preferences...>
      </service>
      <connection id="4">
         <...connection QoS preferences...>
      </connection>
      <connection id="5">
         <...connection QoS preferences...>
      </connection>
   </service>
</partPrefs>
```

Figure 42. Example messages for the translation of QoS preferences

Figure 43 shows example messages exchanged between the EM and a composite Supplier for combining QoS estimates. These messages are exchanged after the EM identifies the best match for the request in Figure 39, and computes the estimated QoS for all the Suppliers directly supporting the user's task. In the example shown, the QoS estimates for the parts of the *speech-to-speech* service are included in the `combineQoS` message. To perform the combination of estimates, the composite Supplier uses its knowledge about the mapping between the QoS dimensions of the parts and each dimension in the QoS preferences for the composite service. The overall QoS estimate is included in the `wholeQoS` reply. For reducing the message traffic during a sequence of `budget` or `setup` requests, the EM may cache the QoS estimates for all the Suppliers, and invalidate the estimate for the composite Supplier when any of the parts is adjusted – and only then request a new combination of the updated estimates.

```
<combineQoS location="Acm-IPaddr" bind="Acm-PortForEM"">
   <service type="speechToSpeech">
      <service id="1">
         <...speech-recognition QoS estimate...>
      </service>
      <service id="2">
         <...language-translation QoS estimate...>
      </service>
      <service id="3">
         <...speech-synthesis QoS estimate...>
      </service>
      <connection id="4">
         <...connection QoS estimate...>
      </connection>
      <connection id="5">
         <...connection QoS estimate...>
      </connection>
   </service>
</combineQoS>

<wholeQoS location="Acm-IPaddr" bind="Acm-PortForEM"">
   <service type="speechToSpeech">
      <...speech-to-speech QoS estimate...>
   </service>
</wholeOoS>
```

Figure 43. Example messages for combining QoS estimates

8 Discussion and future work

The ultimate goals of this research are to demonstrate that, first, the automatic configuration and reconfiguration of Ubicomp environments can increase the *benefit* to users, relative to traditional systems. And second, such automatic configuration can be supported by exploiting lightweight descriptions of user tasks.

The immediate challenges that stem from these goals are: what are adequate semantic primitives to describe the user's task and intent? What degree of sophistication in such descriptions will optimize the utility (benefit vs. cost) for the user? Which functionality to incorporate in the infrastructure for capturing task descriptions; for automatically configuring the environment; and for explaining the configuration decisions to the user? How should that functionality be structured in order to address the Software Engineering issues of building Ubicomp infrastructures and applications? Which are adequate metrics to evaluate the user's costs and benefits associated with configuring Ubicomp environments, under scenarios of user mobility, everyday computing, and environment change? Which are representative scenarios in those categories?

This report addresses the challenges of (a) defining semantic primitives to describe the user's task: user-level state of a task and a utility framework for expressing user preferences and intent. And (b), defining groups of functionality, assigning them to architectural layers, and clarifying which assumptions are shared across those layers.

As part of the Aura research, future reports will present the detailed design of the architectural components in Figure 4. Specifically, we will describe on the design and implementation of Prism, including the functionality for defining user tasks, as well as for explaining the infrastructure's actions. We will also describe a framework for evaluating the utility of automatic configuration for the user, and conduct an evaluation against a set of representative scenarios.

9 References

1. G. Abowd, E. Mynatt. Charting Past, Present and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction*, 7(1), pp 29-58, March 2000.
2. R.K. Balan, J.P. Sousa, M. Satyanarayanan. Meeting the Software Engineering Challenges of Adaptive Mobile Applications. Carnegie Mellon University Technical Report, CMU-CS-03-11, February 2003.
3. S.W. Cheng et al. Software Architecture-based Adaptation for Pervasive Systems. International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing. Karlsruhe, Germany. *LNCS* Vol. 2299, Schmeck, Ungerer, Wolf, (Eds.) April 2002.
4. E. Dashofy, D. Garlan, A. Koek, B. Schmerl. xArch: an XML Standard for Representing Software Architectures. http://www.isr.uci.edu/architecture/xarch/
5. D. Garlan, R. Monroe, D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, Leavens and Sitaraman (Eds), Cambridge University Press, pp. 47-68, 2000.
6. D.Garlan, D.Siewiorek, A.Smailagic, P.Steenkiste. Project Aura: Toward Distraction-Free Pervasive Computing. IEEE Pervasive Computing, April-June 2002.
7. J. Magee, J. Kramer. Concurrency, State Models & Java Programs. John Wiley & Sons, 1999.
8. J.P. Sousa. Relieving Users from the Distractions of Ubiquity: a task-centered architectural framework. Thesis proposal, Carnegie Mellon University, December 2, 2002.
9. J.P.Sousa. Online specifications of the Aura Software Architecture. www.cs.cmu.edu/~jpsousa/research/auraSpecs
10. M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, pp 94-100, September 1991.