A Distributed Object Framework for Pervasive Computing Applications

by

Hubert Pham

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

Author .....
Department of Electrical Engineering and Computer Science
June 2005

Certified by .....
Stephen A. Ward

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Distributed Object Framework for Pervasive Computing Applications

by

Hubert Pham

Submitted to the Department of Electrical Engineering and Computer Science
on June 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a new architectural abstraction for developing dynamic and adaptive software. Separating application logic from implementation mechanism provides developers with a simple API for constructing new application functionality by connecting together a set of generic, distributed software modules. Developers codify adaptive application structure and logic in a simple, synchronous environment, and use the API to control and monitor the resulting implementation of highly parallel and asynchronous module networks. The design and implementation for this architectural abstraction is embodied in the *Resources* framework, a language- and platform independent software component platform geared for pervasive computing application development.

Thesis Supervisor: Stephen A. Ward
Title: Professor

Acknowledgments

I am deeply grateful to my advisor, Steve Ward. He has guided me at every junction in both this project and my graduate career, providing endless encouragement, inspiration, and humor. I couldn't have asked for a better teacher.

I also thank the O_2S family. Umar Saif helped me understand the context for our work and always engaged me to seek interesting problems. I am grateful for my officemate and friend, Justin Mazzola Paluska, who meticulously reviewed early drafts and offered his LaTeX prowess. Also, Eugene Weinstein generously provided his expertise throughout with the Galaxy speech recognition system. A special thanks goes to our Cambridge collaborators, especially Daniel Gordon, for taking on the herculean task of porting this project to C.

I am fortunate to have wonderful friends who have offered perpetual encouragement and moral support: Mariya Barch, Kathryn Chen, Laura Elliott, Lee Lin, Erica Peterson, Mauli Shah, Adrian Solis, Jason Waterman, Brett Whittemore, and Amy Williams. I also extend special thanks to Max Van Kleek – colleague, mentor, and friend – for his guidance throughout the years.

Finally, I am indebted to my parents. They have sacrificed much to afford me this amazing opportunity and privilege. For this, I am deeply grateful, and I love them very much.

Contents

List of Figures

List of Tables

Chapter 1

Motivation

While it is now universally acknowledged that computers enhance our lives in immeasurable ways, it is also universally lamented that these computers also add considerable complexity to our already complicated lives.

Present-day user interfaces are often unwieldy and require significant user training; personal data is scattered across multiple devices and disparate systems, yielding little control to the user; and worst yet, computer failures often disrupt and frustrate users, providing little by way of recovery. Due to the ineptness of computers to function fluidly in the user's world, users often must compensate by learning to operate in the computer's world [1]: for instance, by today's standards, a student's education is grossly lacking without some basic literacy in these computer interfaces [2]. In a sense, today's students must now gain proficiency in managing the complexity that computers have already injected into their lives.

1.1 The Wonders of Pervasive Computing

Weiser writes [3] that "the most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." Weiser's insight suggests that a new rising trend in personal computing, pervasive computing, promises to undo the burdens that the information age has brought upon us.

Pervasive computing attempts to reverse the roles between humans and computers, which have traditionally forced users to work in the computer's world. In the near-future, computation will be ubiquitous, human-centered, freely available, and completely interoperable, helping users achieve more by allowing them to worry less [4]. In other words, the promise of pervasive computing suggests that because computers will be both everywhere and human-centric, users will hopefully *notice* them less — along with the usual complexities that plague users today. But if computers are going to be everywhere, they had better also learn to "stay out of the way" [5]. Weisner suggests that in order for pervasive computing to succeed, the technology must ultimately instill a sense of human calmness and user control within computation-dense environments – such that to the user, computation demands less attention, disappearing altogether into the fabric of everyday life.

1.2 The Challenges of Pervasive Computing

If pervasive computing is to be successful at hiding the complexities of computer inter-operability from the users, *where* exactly is pervasive computing to hide it?

The art of developing applications for user-centric, pervasive computing platforms is itself a complex endeavor. Since pervasive computing itself spans across many fields in computer science, software developers may soon find developing pervasive application with traditional tools to be a daunting task: making life easier for the user might mean making life much harder for application developers.

Even while traditional software tools for developing standard distributed applications are highly evolved, and various hardware component technologies (e.g., laptops, hand-helds, wireless communication, mobile phones) exist today, developing pervasive computing applications is still incredibly complex. The problem lies in the fact that the whole is much greater than the sum of its parts [6]: the difficult challenge here is achieving seamless software and hardware component integration into a meaningful, coherent system.

1.3 Pervasive Computing Requirements

MIT's Project Oxygen [4] and similar projects envision a day when users will no longer need to carry their own devices with them; instead, generic devices will be both embedded in the environment and dynamically configurable to bring computation to the user, whenever and where ever she may need it. This vision suggests that pervasive systems must be both adaptive and goal-oriented.

Adaptive

Pervasive computing systems, like that of Project Oxygen, immerse their users in a triad of sensors, invisible servers, and mobile devices that work together to satisfy user requirements: users describe their intent to the computer, and leave it to the system to carry out their will by exploiting the facilities available. One characteristic of such goal-oriented systems is that they must be both adaptive and self-managing: they must be able to continuously monitor changes in user locations and needs, respond both to component failures and newly available devices, and maintain continuity of service as the set of available resources change.

Goal-Oriented

However, conventional techniques for constructing distributed applications, in which a top-level function is decomposed into statically-partitioned sub-functions, each affixed to a particular API, makes such adaptation exceedingly difficult to program. Adaptation in a pervasive computing environment requires planning at a macro-level, possibly involving a wholesale re-structuring of the application. If there is a change in available resources or user priorities, it is often insufficient simply to reconsider how to implement the function specified in each API: it is necessary to reconsider the reason that API was selected, and whether an alternative function and API has now become more appropriate.

A more promising approach to achieve adaptiveness is to have the user express their requirements as an abstract high-level goal, and then let the system automat-

ically satisfy this goal by assembling an implementation that utilizes the resources currently available to the user. The high degree of dynamism in the environment requires that the resolution of a goal not be a static one time process.

1.4 The O_2S Approach

The O_2S System [7] is an environment framework that subscribes to the goal-oriented approach. The O_2S paradigm involves separating the policy (the *goal*) from the mechanism (how these goals are satisfied); this approach is better suited for sustaining users' intent within the highly dynamic nature of pervasive systems. Because the policy is responsible for maintaining user intent, the policy benefits in being divorced from the implementation mechanism, which may often change to suit the environment. Hence, to maintain user intent, the policy must continuously monitor the environment and respond opportunistically to changes in connectivity and device availability. Separating policy and mechanism enables the policy to restructure the implementation, sustaining the high-level goal in response to changing conditions.

Through the separation of policy and mechanism, the O_2S system is roughly divided into two sections: the goal-planning engine and the component system that implements these goals.

1.4.1 O_2S Planning Engine

The O_2S Planning Engine [8] takes an under-specified goal, or intent, and attempts to automatically generate the best strategy to satisfy that goal, given available resources and policies. Once the O_2S component system constructs the implementation from the Planner's strategy, the Planner monitors the state of the pervasive environment. If the set of available resources changes, the O_2S Planner re-evaluates the implementation strategy and revises it as necessary to maintain or upgrade the satisfaction for the original goal.

1.4.2 O_2S Component System

The component system is responsible for constructing the implementation from the Planner's generated strategy, thereby representing the "mechanism" half of the O_2S system. The component system is unique in that it provides the Planner with a novel abstraction, one which also promotes the separation of policy and mechanism.

1.4.3 Thesis Scope

This thesis is focused primarily on exploring the characteristics and benefits of a component system abstraction that separates mechanism and policy.

1.5 Component Architecture Requirements

This section outlines several architectural requirements for an adaptive component architecture and discusses why achieving adaptiveness with traditional distributed network object platforms is a difficult problem.

1.5.1 Simplified Development

Traditional, asynchronous distributed systems are generally too complex and require the developer to be deeply aware of the intricacies for the underlying system and platform. Furthermore, the debugging process for these traditional systems is frustrating at best.

Building adaptive systems with these traditional systems is also difficult because these systems tend to impose a static API between distributed components. The interface is determined at compile-time and provides no mechanism for changing the relationships between these components during runtime to adapt to hardware upgrade or failure.

By separating policy from mechanism, one architectural requirement arises: a separation between the programming interface and the implementation technology. The component interface must be clean and simple, so that the application logic may

modify or construct new implementations when adapting to changing environmental requirements. While the component implementation may be highly parallel and asynchronous, a simple interface enables developers to construct, monitor, and debug these implementations.

1.5.2 Platform Independence and Portability

Many traditional architectural systems are designed only for one language or platform, thereby effectively restricting the set of implementation technologies that can facilitate adaption.

In practice, the implementation technologies employed in a pervasive environment span many different platforms and languages. Since the interface presented to the application logic is abstracted from the underlying implementation, the interface must be completely platform and language independent in order to fully capitalize on the wide variety of implementations available.

A platform-agnostic framework further simplifies the development environment for two reasons. First, a platform independent interface eliminates the need for developers to be aware of platform or language intricacies pertaining to the implementation. Also, the application logic that makes use of this programming interface becomes highly portable.

1.5.3 Efficiency

Both the interface and the implementation must be efficient. The interface should promote efficiency through code reuse, enabling applications to adapt by re-configure the overall implementation using basic, reusable implementation modules. Once the application constructs the implementation, the implementation must be efficient performance-wise to process high-bandwidth data streams.

1.6 Architecture: The Component Abstraction

The thesis explored in this work is that there exists an abstraction that simplifies the process of developing adaptive, distributed systems. While there are a multitude of ways to use existing distributed object packages to fulfill the architectural requirements (see Section 5.1), this thesis explores one abstraction that promotes a separation between policy and mechanism and presents an fitting implementation. We believe that an abstraction focused on this separation effectively simplifies the process of developing adaptive, distributed systems.

Three important features characterize the abstraction. First, the abstraction presents the developer with a simple API and environment, thereby simplifying the process of codifying the policy and application-specific logic. Second, developers use the simple interface to construct the desired implementation by connecting together a set of distributed software modules from a universe of generic components. Finally, while separating mechanism from policy may sacrifice performance for flexibility, the performance cost does not debilitate the component layer implementation. Figure 1-1 illustrates the O_2S component abstraction.

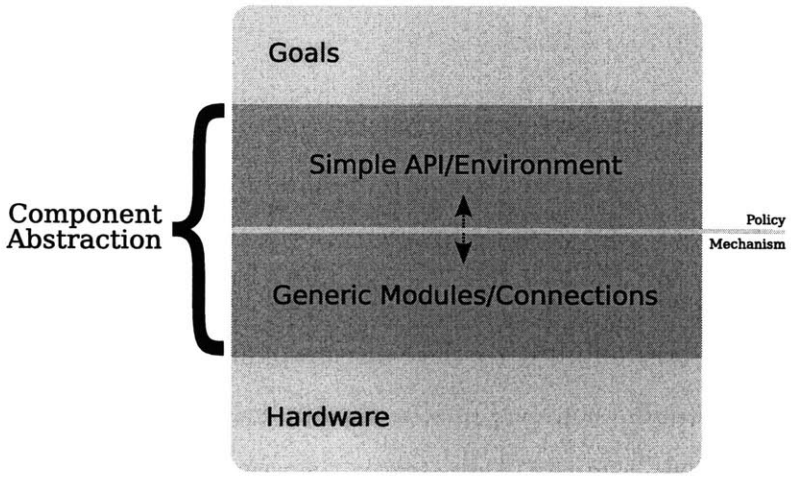

Figure 1-1: The O_2S component abstraction.

1.6.1 Clean Interface

The basic interface provides a mechanism for instantiating a collection of components on various hosts and interconnecting them into a network. The result implements a specific application or functionality; this mechanism promotes a circuit-diagram approach to application construction. Application logic also monitors the operation of the resulting circuit via a stream of high-level messages generated by the components. These message streams, or events, are used to report component failures, user inputs, or various resource-specific notifications. The health of devices hosting these components (and the communication paths between them) is also transparently monitored; component state updates and debugging output are collected, filtered, and serialized for presentation to the application logic.

A clean abstraction simplifies the design, programming, and maintenance of distributed and adaptive applications. The interface encourages developers to focus on the simple, sequential model for high-level application logic, while the compute-intensive reflexive implementation is managed largely automatically. It becomes very natural to express the necessary logic behind adaptive applications, as the interface frees the developer from the implementation details that often complicate the model.

Circuit-Diagram Model

Figure 1-2 is an example of an application that brings a voice recognition service to a user's hand-held, using the abstraction's programming interface. The application first obtains a handle to the voice recognition service and configures the service to send all recognition messages back to the application's message handler. The application proceeds to obtain handles to the voice recognizer's audio input and the hand-held's microphone (audio source); finally, the application simply connects these two audio streams together. Figure 1-3 illustrates the resulting implementation composite, which runs autonomously and sends high level events (e.g., recognized tokens) back to the application logic for processing.

Using this interface, generating new implementations resembles wiring together

circuit elements. This example illustrates how the overall function and intent of application logic becomes transparent when implemented using the abstraction's simple interface.

```
def setup_voice_rec():

  # ask the system to find a voice recognizer service
  voice_rec = system.lookup("Voice Recognizer",
                            grammar = "voice_shell")

  # ask the system for an event queue and specify a
  # method to handle incoming messages
  event_queue = system.get_event_queue(handler = handle_event)

  # instruct the voice recognizer to send recognized tokens
  # back to our event queue
  voice_rec.set_recognition_target(event_queue)

  # ask the voice recognizer for an input connector to where
  # audio waveforms will be streamed for recognition
  voice_rec_input = voice_rec.get_audio_sink()

  # get the audio source from the hand held device
  mic = hand_held_device.get_mic()

  # connect the mic to the voice recognizer
  system.connect(voice_rec_input, mic)


def handle_event(new_event):

  # handle incoming events
  if new_event.recognized_token == "Hello":
     ...
```

Figure 1-2: An example distributed application using a clean abstracted interface.

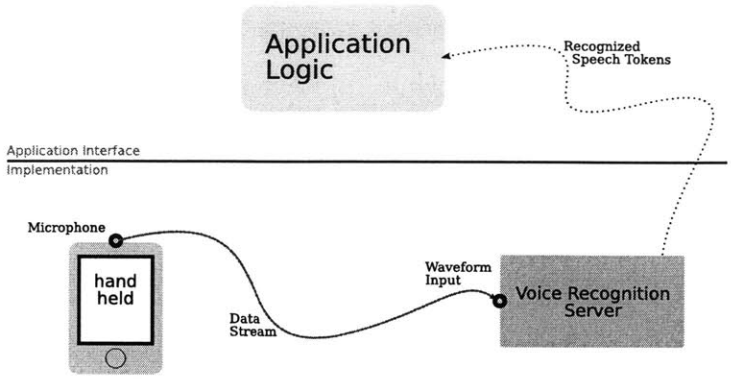

Figure 1-3: The circuit-diagram model for the distributed application.

1.6.2 Reusable, Distributed Modules

Goal-oriented programming in a pervasive computing environment involves dynamic assembly, and subsequent re-restructuring, of available distributed components. Therefore, this abstraction dictates that individual components must be reusable and versatile, suitable for implementing a variety of functionality. The objective is to provide a mechanism that allows a set of components to be selected from a repertoire, including both physical and virtualized components, and interconnected together to implement some application-level service. Furthermore, this abstraction promotes hot swapping of components to upgrade service, as well as controlled but independent evolution of individual code modules.

These code modules run on any platform and are written in any language suitable for the implementation environment; developers manipulate these components through the simple, platform- and language-independent API without concern to the implementation details.

1.6.3 Efficiency — Where It Matters

The simple API gives rise both to rapid application development as well as performance efficiency. First, the universe of generic, distributed code modules promotes code reuse for constructing or adapting implementations, thereby accelerating the development process.

By separating the implementation from the programming interface, this abstraction achieves an advantageous efficiency balance. In providing a clean, synchronous interface for constructing application implementations, the abstraction does sacrifice efficiency for flexibility. However, in the spirit of Amdahl's Law [9], the greatest speedup is achieved by optimizing efficiency over computation that contributes to most of the overall task. The flexibility attained in a simple interface is worth the cost of inefficiency, since this interface is only used to construct and monitor component networks. In turn, it is the components and their connections that constitute most to the overall computation; by separating the application logic from the underlying implementation, these connections are as fast as the underlying operating system socket implementation.

Chapter 2

System Architecture and Design

This chapter first describes the system architectural design of O_2S and discusses how the design addresses the component abstractions presented in the previous chapter.

The chapter then details the key design abstraction in this project, *Resources*. The *Resources* abstraction is used to design and implement several system-level components that compose the O_2S component platform.

2.1 System Architecture

To realize a system that embodies the traits of the abstraction discussed in the previous chapter, the following architecture consists of a layering of interoperable software constructs, depicted in Figure 2-1.

Synchronous Control A standard network object model (e.g., Remote Procedure Call) provides a synchronous control layer, which forms the basis of the simple API and environment for instantiating and connecting distributed modules. Furthermore, the network object model provides the veneer of a simple, sequential, and localized interface for controlling and monitoring the parallel, distributed component networks.

Data Streaming The data streaming mechanism connects components together into a highly parallel, distributed system of interconnected components. These stream connections bypass the Remote Procedure Call (RPC) system and hence do not incur the overhead in the standard, synchronous RPC mechanism. Stream connections are designed for applications that depend on routing real-time or rich media data between distributed modules for processing. These stream connections also encourage component re-use by providing the mechanism for connecting together generic components in ways that form new applications.

Serial Event Stream To monitor errors or other events generated by either the stream connections or the network objects, an event notification system provides a mechanism for sending serial messages to any network object's event queue.

Resource Discovery & Health Monitoring Servers hosting network objects can often fail from network, power or hardware failure. For pervasive environments, it is important that the system detects such failures and inform the appropriate dependencies of the failed network object. In addition to failure detection, the architecture also provides resource discovery, enabling applications to potentially recover from failures by discovering and substituting the failed object for

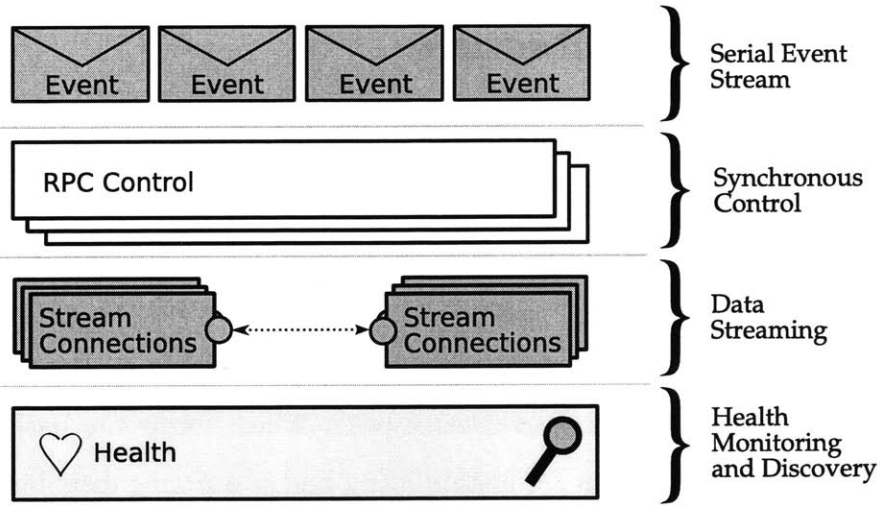

Figure 2-1: The layering of software constructs that meet the system architectural requirements.

an alternative resource during run-time.

2.2 The *Resources* Abstraction

The *Resources* abstraction is a versatile Remote Procedure Call [10] (RPC) framework that greatly facilitates the system design. This section describes the *Resources* framework and serves as a preface to the System Design and Model (Section 2.3).

2.2.1 Background: The Standard RPC Model

Traditional RPC is based on a client/server model. Hosts designated as "servers" provide computational services to "client" hosts (see Figure 2-2).

Typically, developers using RPC define the interface for their services and then generate server and client stubs. Calls made to remote services are redirected to these stubs, which provide the machinery for the network communications and data marshalling (Figure 2-3).

2.2.2 The *Resources* Model

RPC is an appropriate paradigm for developing pervasive computing applications, which may span across many distributed devices, and *Resources* provide an additional abstraction layer for developing and using RPC objects.

A *Resource* is an abstract object, with methods and state. Specifically, *Resources* are *network* objects, in that they can be passed between hosts and processes, and (remote) hosts can invoke synchronous method calls on these *Resources*. *Resources*

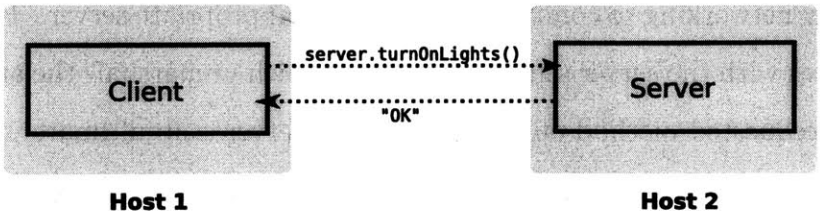

Figure 2-2: The typical RPC Promise.

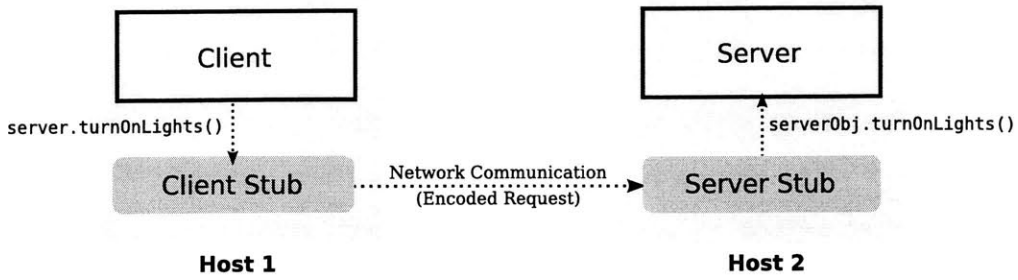

Figure 2-3: The typical RPC Reality. The server's return value traverses the reverse path outlined by the arrows.

---

are abstract in the sense that they hide the specifics of the underlying RPC implementation, while presenting a simplified and universal semantic for method calls on all objects. One objective of *Resources* is to alleviate the distributed application developer from the burden of varying interfaces and semantics between remote and local objects.

The *Resource* idiom is similar to that of object oriented programming; *Resources* provide a framework for bundling a set of coherent services or computational resources into a modular, network object.

2.2.3 Dynamic Stub Generation

Most RPC implementations involve generating "stub" code for client and server objects. These stubs serve to interface the developer's remote procedure calls to the underlying mechanism that implements the remote procedure call.

For instance, clients that invoke a remote procedure call actually invoke a local call on the client stub. This client stub typically then marshals the arguments and invokes the necessary networking to communicate with the appropriate server. The client stub communicates with the server stub, the latter of which unmarshals the arguments and invokes the requested method on the server. The server stub ultimately marshals the return value and sends the result to the client stub. (Figure 2-3).

Unfortunately, such traditional RPC schemes (detailed in Section 5.1) place unnecessary and burdensome tasks on the developer. Not only must developers generate

client and server stubs for each code module, but even before that, the developer must also be cognizant of where (which physical hosts) the code modules will execute, as well as plan *a priori* how their code modules should best span different hosts. The amount of manual effort and planning often renders traditional RPC systems unwieldy for implementing dynamic, pervasive computing environments.

Resources address the problem of stub generation by automatically generating the necessary stubs at runtime. Developers simply subclass a base-class provided by the framework and develop their server objects without any special consideration to stub generation. During execution, if a *Resource* instance is passed to remote objects, a client stub for that *Resource* instance is dynamically generated and marshalled across the network to the remote object. The client stub, which on the remote server represents the original server *Resource* instance, intercepts the designated methods on behalf of the server *Resource* instance and provides similar functionality as a standard client stub discussed above.

On the other hand, if the *Resource* instance is passed to a local object (i.e., an object instance running in the same address space), no client stub is generated; instead, a standard reference to the *Resource* object is passed. All calls on the *Resource* object reference are hence local calls and do not incur the overhead of marshalling and network latency.

Finally, if *Resource* stubs are eventually passed back to the host with the running *Resource* instance, the stub is converted back into a local reference to the *Resource* instance. This "interning" effectively enforces the invariant that stubs are only generated and used as handles to *Resources* which are remote (with respect to the handle); otherwise, if the *Resource* image is running in the local process, all handles to that *Resource* are local memory references, thereby requiring no network communication.

This scheme relieves the developer from generating stubs for her code. Furthermore, the scheme unhinges the need to know *a priori* the execution location for code modules, since pervasive environments often determine such parameters dynamically during runtime. The great advantage is that the developer need not be aware of whether procedures are implemented locally or remotely: the invocation API is stan-

dardized, and the optimal invocation mechanism is always automatically executed for all objects, local or remote.

2.2.4 Naming

As developers create *Resource* code modules, they will need some mechanism for naming these *Resources* to promote code reuse among developers, as well as laying the foundation for a (dynamic) lookup service for code modules.

To standardize and formally capture the characteristics of a *Resource* class, each *Resource* class is associated with a document containing an immutable description of that *Resource*. The URI location of this document serves as the Resource class's unique type, which effectively identifies and names the *Resource* formally.

In general, this description will contain a mix of formal interface specifications (method signatures, etc.), informal descriptions (of the sort found in man pages), and other potentially useful information including code for test cases and demonstrations. This target description, encoded in XML, will serve a role analogous to that of WSDL descriptions for web services and may use similar mechanism.

2.2.5 Typing: *RType*

There are a number of supported data types that may be passed as parameters between *Resources* instances on different hosts. These supported data types are known as *RTypes*; the *RType* interface standardizes the serialization of marshallable data parameters. By default, *RTypes* include basic types such as integers, floats, strings, lists, booleans, key/value maps, as well as *Resource* objects. For *Resource* types, stubs for the *Resource* are automatically generated and serialized.

Furthermore, *RTypes* are extensible: the developer may also extend *RTypes* to include arbitrary objects, by supplying the necessary serialization procedures for her custom objects.

Serialization

All *RType* object instances are passed by copy, except for *Resource* objects. *Resources* are essentially passed by reference via automatic stub generation. These client stubs (handles) store a link to the running image of the (server) *Resource*; the client stub intercepts method calls to *Resource* and fields these calls to the server for execution.

In other words, if non-*Resource RTypes* are passed as parameters or return values for a remote call, the values of these variables are copied across address spaces. For *Resources*, on the other hand, only one instance of state for that *Resource* exists, regardless of how many *Resource* handles exist on remote servers. The state on the server can potentially be modified by a remote procedure call from any of the handles.

Wire Encoding Format

RTypes serve as a data typing abstraction layer above the underlying RPC implementation. Developers treat *Resources* as first class data types, and pass references to these objects to remote hosts without concern to the specifics of any chosen RPC implementation. In a sense, *RTypes* encodes a rich and extensible selection of data types (most notably *Resources*) into the underlying RPC implementation *du jour*.

After serialization, *RTypes* are ultimately encoded into a language- and platform-neutral wire format. This encoding format consists primarily of basic types generally supported by virtually all standard RPC implementations. The *RType*'s agnostic property forms the foundation for the underlying implementation of language- and platform-independent *Resources*.

2.3 System Design and Model

The *Resources* abstraction described above facilitates the design of a component system that fulfills the architectural design outlined in Section 2.1. This section describes the specific O_2S component system design, which addresses each architectural layer depicted in Figure 2-1.

2.3.1 Overall Picture

Figure 2-4 illustrates the relationships between the system components described below.

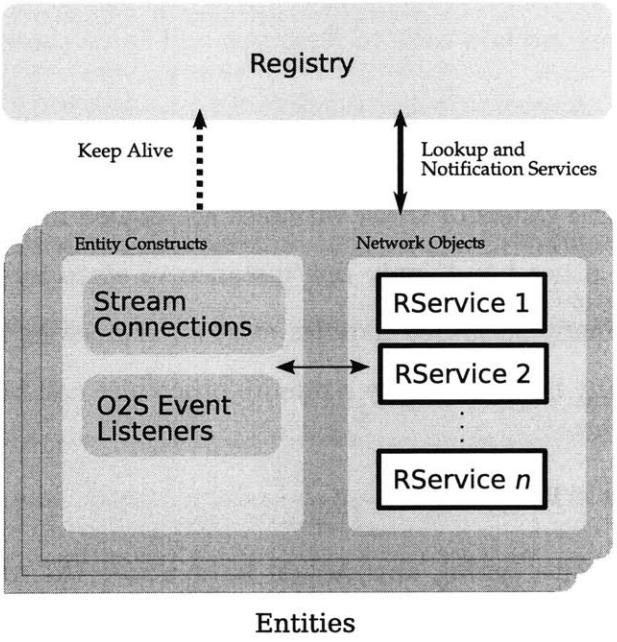

Figure 2-4: The system components that fulfill the system design requirements.

2.3.2 RService Network Objects

The `RService` represents the network object implementation, using the *Resources* architecture. The `RService` fulfills the network object requirement of the architecture. To develop RPC server objects, developers simply subclass the `RService` object class and add instance methods. In addition to implementing these instance methods, developers must also specify which methods may be accessed remotely: these are the network exportable methods of the object, and the collection of these methods form the public API for the object.

Since the `RService` base class is built upon the *Resource* architecture, developers enjoy automatic data marshalling of *RTypes*, automatic stub generation, and so forth.

The `RService` is designed to help developers construct a single, coherent network-exportable object that implements a computation service on the serving host.

2.3.3 Stream Connections

A stream connection supports uni-directional data flow and consists of two connected Stream Connectors. The connectors themselves are designated as either input or output, and a stream connection connects two appropriately gendered connectors. These connectors are used to connect together different modules (e.g., `RServices`) and fulfill the data streaming architectural requirement.

A raw byte data stream that enters an input connector is simply sent across the connection to the output connector. Furthermore, Stream Connectors also support out-of-band data. The connectors are extensible in that developers can subclass the standard stream connector class to format and interpret an out-of-band data stream, which is often useful for meta-data tagging.

Connectors can be "wired" directly to take input or send output data to hardware devices (e.g., a microphone or speaker, respectively), but often, data received from a connector is further processed before becoming input to a different connector. These stream connections form the backbone of a highly parallel and distributed network of component modules. The operation of connectors are somewhat autonomous, in the sense that once the network is established, data simply flows between modules.

To initially set up and control the network flow, the Stream Connectors rely on the *Resource* architecture. Stream Connectors themselves are built upon *Resources*, and therefore references to these connectors can be passed to remote hosts. These references are designed to be passed to hosts running the application logic, which connects the appropriate connectors together to form the desired component network.

2.3.4 O_2S Events

The Event System provides a general and light-weight mechanism for sending asynchronous notifications between *Resources*. The system includes a data structure,

namely an Event, which contains several fixed data fields. These data fields can store any *RType* value; while there are suggested semantics for these fields, they can be used (or ignored) for any application specific purpose. Events are often used to report errors back to application-level logic.

Events are designed to be easily constructed and may be sent (thrown) to any Event Listener on a different host. Event Listeners receive all Events thrown to that Event Listener and either queue the received events or pass them directly to a callback for processing. Event Listeners are services that can be easily instantiated for RServices and in a sense provide the RService with a message loop for processing events from external, remote sources.

It should be noted that Event Listeners are themselves *Resources*, so these listener objects can be passed to remote hosts. Once any host possesses a handle to an Event Listener instance, that host may throw Event structures to the event listener by invoking a method on the Event Listener with the Event structure as a method parameter. In this way, the Event System is simply a construct implemented using the *Resources* framework.

2.3.5 The Entity

Entitys represent the logical host and acts as a container for RServices. Each Entity runs in a separate process and provides an environment for executing and serving multiple RServices on the host. In addition to managing a collection of RServices, Entitys also provide these RServices with an interface to both the system-level constructs described above as well as the outside world – analogous to an operating system which provides (for applications) both an execution environment, as well as an interface to system-level calls and devices.

In developing an application specific RService network object, the developer may need several Stream Connectors, Event Listeners or may need to look up and retrieve a handle to external *Resources*. The Entity that hosts this RService is designed to manage and provide these services by instantiating on-demand new Event Listeners or Stream Connectors for requesting RServices. The Entity also garbage

collects these constructs when they are no longer used.

Additionally, the `Entity` provides its hosted `RServices` with an interface to the rest of the O_2S system at large. This allows `RService` to access an appropriate look-up service for locating other *Resources* (see Section 2.3.6). Also, as a host for `RServices`, `Entitys` often serve as the gateway between the O_2S system and the hosted `RServices`. Because `Entitys` are themselves implemented as a special `RService` (and therefore a *Resource*), a handle to an `Entity` can be passed to remote hosts. This provides remote hosts with a mechanism to access handles for `RServices` via the hosting `Entity`.

When the developer instantiates an `Entity`, she names the `Entity` (with a string) and sets a variety of application-specific meta-parameters that further identify the `Entity` instance. Finally, since the `Entity` serves as an interface between the `RService` objects and the outside world, rather than monitor liveness for each `RService` network object (incurring great overhead), the system is designed to monitor liveness just for the `Entity` process itself. If the `Entity` process terminates – or the host suffers from network failure – then the system can infer that all `RServices` running on that `Entity` become unavailable.

2.3.6 O_2S Registry

The `Registry` provides liveness (or health) monitoring, a notification subscription service, and a *Resource* lookup service. `Entitys` discover their local `Registry` and proceed to register their name and meta-parameters (upon start-up) with the `Registry`; in turn, the `Registry` maintains a database of registered `Entitys` running in the system and monitors their liveness. However, the `Registry` does not participate nor serve as an intermediary in communication among registered `Entitys` (or their `RServices`).

The `Registry` monitors the health of `Entitys` via periodic "keep-alive" UDP tokens. Once an `Entity` is registered with the `Registry`, the `Entity` must send an identifying UDP token to the `Registry` at fixed intervals.[1] If the UDP token fails to

[1]The `Entity` sends UDP tokens to the `Registry`, rather than vice versa, to account for potential

reach the Registry (due to network or Entity host failure) after a few intervals, then the Registry marks the Entity as dead and removes the Entity from the database of live Entitys.

The Registry also provides a notification service based on subscription. Entitys can subscribe with the Registry to be notified whenever a *different* ("target") Entity begins registration or is marked as dead. Entitys can place a subscription with the Registry by specifying either the name and/or a subset of meta-parameters for the target Entity, along with whether notifications should be sent on the target Entity's registration, outage, or both. The Registry sends notifications using the Event system described in Section 2.3.4.

Finally, the Registry provides a simple directory lookup mechanism so that Entitys can find each other. As with most constructs in the O_2S system, Entitys are also *Resourcess*; when Entitys register with the Registry, the Registry also stores a reference to the registering Entity. Other Entitys may then query the Registry for handles to all registered Entitys in O_2S system.

2.4 Reference Tracking & Garbage Collection

For tracking RService handles across hosts, RServices come in two flavors: untracked and tracked.

2.4.1 Untracked RServices

By default, RServices are untracked: when references to untracked RServices are passed to "client" Entitys, no bookkeeping is performed to keep track of which Entitys possess these references. While for most applications there is little need to track the list of Entitys that possess an RService's handle, some applications can be simplified with the availability of RService handle tracking information.

---

firewall issues.

2.4.2 Tracked `RServices`

One such benefit to tracking handles is that `RServices` can easily determine when to allocate and deallocate system state and resources (e.g., GUI resources), which may be a function of the number and location of the handles to the `RService` in circulation.

For tracked `RServices`, an `RService` can ask its hosting `Entity` for a list of "client" `Entitys`, that is, `Entitys` that possess a remote handle to the `RService`. Additionally, tracked `RServices` can also request its hosting `Entity` for a notification whenever the list of remote client `Entitys` changes.

When client `Entitys` (or any `RService` on client `Entitys`) determine that they no longer need a certain tracked `RService`, they can either destroy the tracked `RService` handle, wait for the tracked `RService` handle to be garbage collected by the interpreter or operating system, or they may actively call the `close()` method on the tracked `RService` handle. In all cases, the system will notify the hosting `Entity`, which may in turn notify the appropriate `RService`.

Chapter 3

Implementation

This chapter first describes the underlying implementation techniques for the architecture and system design, followed by a discussion of the implementation technologies used.

The O_2S component system is composed of several implementation layers that implement the system design discussed in Section 2.3. The following sections describe the implementation layers, depicted in Figure 3-1, and discusses how the implementation fulfills the architectural design requirements.

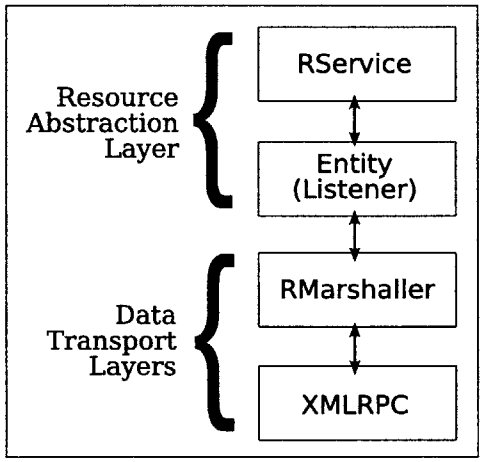

Figure 3-1: The Layers of the O_2S *Resources* system.

3.1 Data Transport

The O_2S component implementation relies on the data transport layer for data marshalling and network communication for remote procedure calls, as shown in Figure 3-2. The `RMarshaller` converts *RType* data types into a language- and platform-independent intermediary representation. This representation is then passed to the RPC Transport layer, which handles the actual RPC communication.

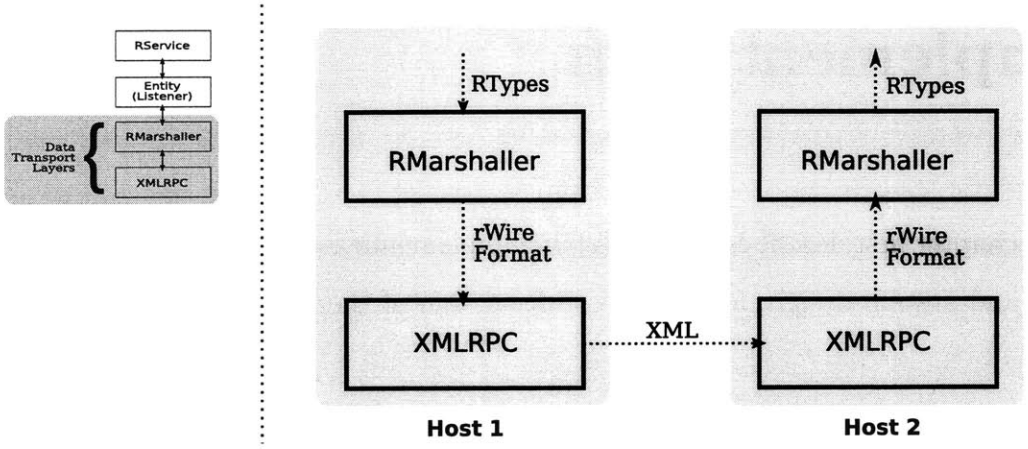

Figure 3-2: The Data Transport Layers of the O_2S *Resources* system.

The data transport layer resembles the OSI model [11]: as data traverses down each layer on one host, the data is further encoded or transformed to adhere to the subsequent layer's abstraction rules. Ultimately the data is sent across the network to the destination host, where the inverse-transformation occurs as the data traverses the layers upwards.

3.1.1 RPC Transport

The lowest layer is the RPC Transport. This layer is implemented with a standard XML-RPC [12] library. XML-RPC is a remote procedure call specification that describes an XML representation for encoding serialized data. XML-RPC uses HTTP as the transport and is designed to be simple and fast; many implementations exist for a variety of languages and platforms.

While the O_2S component implementation requires an RPC implementation, it does not depend on that implementation being XML-RPC. One could easily swap out the XML-RPC implementation for a different RPC; however, XML-RPC is an ideal choice today given its simplicity and transparent, standardized encoding.

3.1.2 Typing and Data Marshalling

The `RMarshaller` (layer) handles data marshalling. This layer accepts *RTypes* and transforms them into a language- and platform-independent intermediary representation.

The `RMarshaller` is necessary in conjunction with the RPC marshaller because the `RMarshaller` must implement the *RTypes* typing abstraction. The data types supported by *RTypes* are generally a superset of the data types supported by most RPC packages: while both *RTypes* and standard RPC marshallers support the fundamental data types, *RTypes* are extensible by design to include custom, developer-designed objects, as well as `RService` objects. In a sense, *RTypes* provide a simple framework and abstraction for developers to define and treat high level objects as first class objects. The `RMarshaller` transforms all *RType* data types to a platform agnostic encoding for the RPC marshaller.

Supported Types and Language Mapping

The `RMarshaller` supports the following default *RTypes*:

- **integers**: four-byte signed integers
- **double**: double-precision, signed, floating point
- **boolean**: `True` or `False`
- **string**: ASCII formatted string of characters
- **list**: finite sequence of *RTypes*
- **dictionary**: finite mapping of *RTypes*

- **None**: a null type
- **RStruct** a dictionary with pre-defined keys
- ***Resource*** an `RService` network object

Since *RTypes* serve as an abstraction layer for the marshallable data types, there must also be a binding between the data types of the programming language and *RTypes*. Given that the O_2S component system is designed to be language independent, and the supported types of *RTypes* are restrictive and standard, implementing bindings for any language is generally trivial. Table 3.1 illustrates example bindings between two languages and *RType* data types.

Table 3.1: Python and Java language bindings for *RTypes*.

RType	**Python**	**Java**[a]
String	`str`	`java.lang.String`
Integer	`int`	`java.lang.Integer`
Boolean	`bool`	`java.lang.Boolean`
Double	`float`	`java.lang.Double`[b]
Dictionary	`dict`	`java.util.Hashtable`[c]
List	tuple	`java.util.Vector`
Resource	`system.RService`	`edu.mit.csail.o2s.system.RService`
None	`None`	`edu.mit.csail.o2s.system.utilities.Null`[d]

[a]Compatible with Java 1.4 onward; Java 1.5 supports auto-boxing of data types, but the current implementation does not rely on this feature. The system will marshal primitive data types and convert them to the the object equivalent.

[b]`java.lang.Float` is also supported.

[c]Future implementations will support any object implementing the `java.util.Map` interface.

[d]`null` is also acceptable.

Serializing Objects

The `RMarshaller` simply defines an interface for marshalling *RTypes* described above. The following discussion describes a specific `RMarshaller` implementation that optimizes for a transport layer that employs XML-RPC. If XML-RPC is swapped out for a different RPC technology, the system developer simply re-implements the `RMarshaller` to serialize the *RTypes* with a suitable encoding for that RPC package.

The term "`RMarshaller`" from here onwards refers to the specific XML-RPC implementation of the `RMarshaller` interface.

Scalars and Lists Serializing scalar *RTypes* (integers, strings, floating point, boolean, and null types) is considerably simple since these data types are also supported by XML-RPC. Hence, if the `RMarshaller` is asked to marshal these data types, it simply passes them directly to the XML-RPC library for wire marshalling.

Since XML-RPC also supports lists, serializing *RType* lists simply entails serializing each element, and placing the serialized representation in an XML-RPC list.

Dictionaries, Resources, and None Types XML-RPC has no built-in support for *Resources*, so the `RMarshaller` encodes several *RTypes* using a XML-RPC dictionary, thereby "multiplexing" *RTypes* into a flexible, supported XML-RPC type.

Because *RType* dictionaries, *Resources*, and None types are all encoded with an XML-RPC dictionary, it is necessary that the XML-RPC dictionary encoding adheres to a special format with predefined keys. The fields to these keys store the meta-data that characterizes the *RType* for future unmarshalling (the *meta_type* fields), as well as the serialized encoding of the *RType* (the *data_encoding* field). The discussion that follows details how different *RTypes* are encoded for storage in the *data_encoding* field of the XML-RPC dictionary; see below for a discussion on the XML-RPC dictionary format and the meta-data fields for these special *RTypes*.

To serialize a *RType* dictionary, the keys and values for each key/value pair is serialized by the `RMarshaller`. The resulting serialized pairs are then aggregated into an XML-RPC list, which then serves as the serialized encoding for the *data_encoding* field.

`RServices` objects are serialized by storing enough information to construct a client-stub to the `RService` object. The `RMarshaller` achieves this by first storing this information in an *RType* dictionary; the serialized encoding of this dictionary then serves as the encoding for the `RService`. Among the necessary information stored for stub construction includes: the address and port of the XML-RPC server,

the names and signatures of exported RPC methods, as well as various routing metadata. See Section 3.2.1 for an in-depth discussion.

To serialize None types, the *data_encoding* field is simply the empty string.

Custom *RTypes* Since *RTypes* are extensible by the developer, the `RMarshaller` also must support custom, developer-defined *RTypes*. The serialization of custom *RTypes* is also multiplexed into the XML-RPC dictionary encoding.

When the developer implements the custom *RType*, the developer must adhere to the *RType* interface by supplying the necessary marshalling (and unmarshalling) procedures to serialize state for the custom *RType* object. The `RMarshaller` simply calls the custom *RType*'s `marshal()` method to obtain the (intermediate) serialized state; the `RMarshaller` then serializes this intermediate encoding into the appropriate XML-RPC form and stores the result into the *data_encoding* field of the final XML-RPC dictionary encoding.

(Remote) Exceptions The developer may often need to throw exceptions in the `RService`, so these special error objects must be passed back to the (remote) caller. The O_2S system defines and implements several different exception classes for use with *Resources*. In the current implementation, if an exception occurs, the exception is encoded using the standard XML-RPC Fault (exception) objects (with a string and traceback to denote the exception cause). These Fault objects are automatically converted into the implementation language's default exception type when unmarshalled. This scheme alleviates the need to introduce yet another *RType* for exceptions, but the introduction may become necessary in the future if developers need to classify remote exception objects.

rWire Format

As mentioned above, the wire format for standard *RType* scalars and lists have direct analogues in XML-RPC. However, for *RType* dictionaries, `RServices`, None types, and custom *RTypes*, the `RMarshaller` multiplexes the XML-RPC dictionary to en-

code these advanced types. The encoding standard for this XML-RPC dictionary requires the keys listed in Table 3.2.

Table 3.2: XML-RPC dictionary encoding format for advanced *RTypes*.

	Key	**(Field) Value**
meta_type	`type`	Describes the *RType*
	`spec`	XML specification for the *RType*
	`reconstruction_parameters`	Reconstruction data for custom *RTypes*
data_encoding	`value`	Serialized encoding of the *RType*

The `type` key specifies the *RType* data type serialized. It can take on the field values: `Dictionary`, to specify that the serialized *RType* is a dictionary; `Resource`, to specify a serialized `RService`; `NoneType` for None types; and `Arbitrary` for custom *RTypes*. These field values enable the `RMarshaller` to determine the data type encoded in the `value` field, essential for the unmarshalling process.

The `spec` key specifies a URI pointer to an XML document that describes the type.[1]

For custom *RTypes*, the `reconstruction_parameters` provide pointers to the location and class names for the developer-defined custom *RType* code implementation. Currently, the implementation assumes that the developer deploys the implementation for her custom *RTypes* on all hosts that can potentially accept and unmarshal instances of the custom *RType*. Future versions may use the `reconstruction_parameters` field to specify a URI pointer to signed code implementations for the custom *RType*.

Finally, the `value` field holds the serialized encoding of the data type as discussed above.

[1]The `spec` key is designed to ultimately fulfill the role of the `type` key discussed above; as of writing, the specification locations were not ready, so the `type` key provided a temporary implementation.

3.2 *Resource* Abstraction Layer: The `Entity`

The *Resource* network objects, or `RServices`, provide another level of abstraction above the network data typing and transport layers. The developer usually interacts with the O_2S component framework at the *Resource* abstraction layer, through the process of writing `RService` network objects and installing them within `Entitys`.

3.2.1 *Resource* Network Objects

As discussed in Section 2.3.5, the `Entity` represents a logical host machine and provides an environment for running `RServices`. Figure 3-3 illustrates the *Resource* abstraction layers, which spans the internal architecture of `Entitys`.

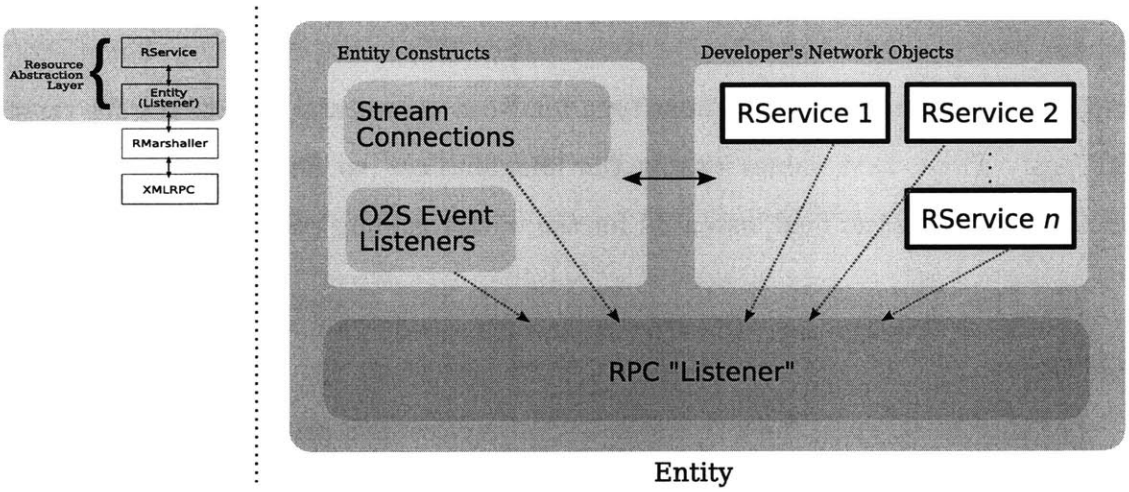

Figure 3-3: The Resource Layers of the O_2S *Resources* system.

RPC Multiplexing: `Listener`

At the heart of every `Entity` is one `Listener`. The `Listener`'s role is to serve as a "switchboard," intercepting all incoming RPC requests and fielding these requests to the appropriate `RService` for handling.

The advantage of this architecture is that the system avoids the large networking overhead of running an independent XML-RPC server for each `RService` that is instantiated on the `Entity`. By multiplexing a single XML-RPC server (managed by

the Listener), the overhead incurred by adding additional RServices to the Entity is constant.

Nonce

The Listener runs one XML-RPC server and performs the necessary bookkeeping to properly field incoming requests to the correct RService. To do so, RServices are all assigned a nonce[2] that is specific and unique to each RService. The nonce take on a specific format that guarantees its uniqueness among all RServices in the O_2S system, thereby identifying a specific RService instance. Figure 3-4 illustrates an example of an RService nonce.

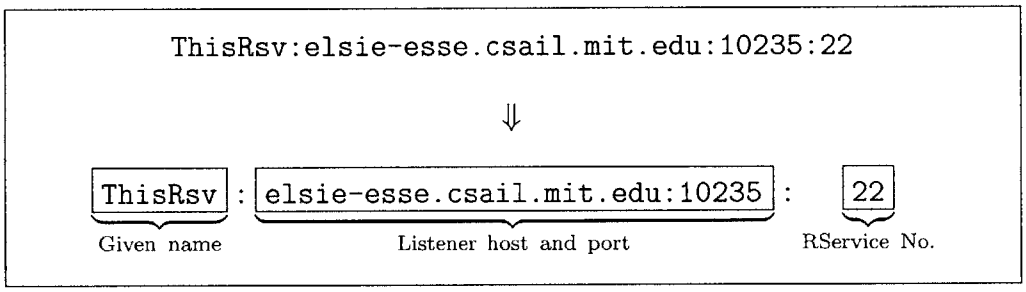

Figure 3-4: An example of an RService nonce.

The fields of the nonce include:

Given Name The given name is the name given to this class of RServices by the developer. The name distinguishes the class of the running RService instance.

Listener Host and Port This element specifies the location (hostname and port) of the XML-RPC server, managed by the Listener. These parameters distinguish the physical location of the running RService image.

RService Number The Listener assigns an RService Number to each RService hosted by the Entity. The RService Number is unique to each RService instance on the Entity, thereby distinguishing RService instances on any given Listener.

[2]The term "nonce" here takes on a less specific meaning as that which is defined in RFC2002 [13]. "Nonce" here simply refers to a one-time-use unique string token.

The nonce construction uniquely identifies each `RService` instance running in the O_2S system. The nonce is constructed and assigned to a `RService` when the `RService` is instantiated; during the `RService`'s initialization process, the `RService` must "register" themselves with the `Listener`. The `Listener` assigns a nonce to the `RService` instance, maintaining a table between nonces and `RService` instances.

Client Stubs

When a handle to a `RService` instance is passed to a remote object (often as a parameter or a return value), the `RMarshaller` constructs a serialized encoding of the `RService` that contains sufficient information to generate a client stub (as discussed earlier in Section 3.1.2).

This encoding essentially contains the `RService` nonce. The client stub that is eventually generated on the remote (client) host simply serves as a proxy to the (serving) `RService`, intercepting all method calls to the `RService` from the client host. When the client calls a method on the client stub, the stub forwards the call (along with the marshalled parameters) across the network to the XML-RPC `Listener` on the server (the address of which is encoded in the `RService` nonce). Furthermore, the nonce is always encoded in all remote calls, enabling the `Listener` to ultimately forward the request to the appropriate `RService`. Figure 3-5 illustrates this process.

Object Interning

When `RService` instances register with the `Listener`, the instance is also "interned" to ensure that only one copy of that specific instance exists on the `Entity`. This prevents the scenario where a client stub for some `RService` is passed back to the `Entity` hosting the `RService`. Without interning, both the `RService` and its client stub are instantiated, when in fact they refer to the same semantic object. Furthermore, if method calls are made on the client stub, this would entail the gratuitous overhead of network communication to the same process.

Object interning promotes the convenient abstraction *Resources* provide in a dis-

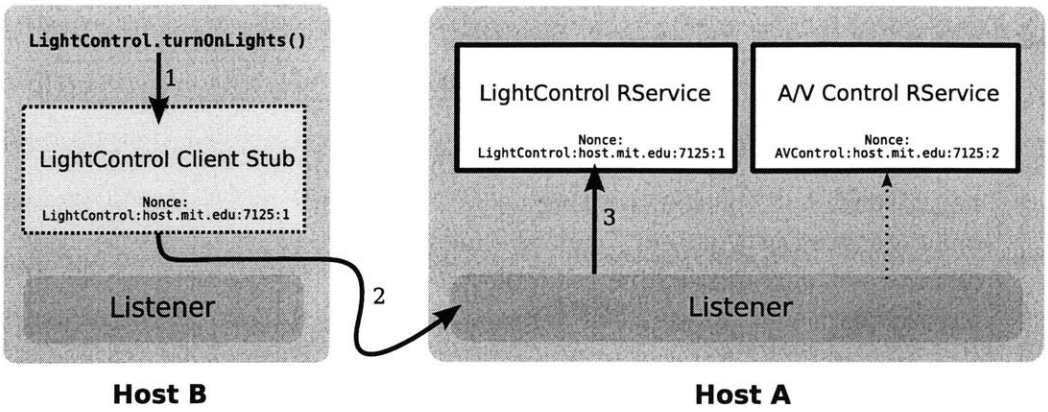

Figure 3-5: The Client Stub in action. The `RService`'s nonce is encoded in the request (Step 2); the `Listener` of Host A forwards the request to the correct `RService` (Step 3), based on that encoded nonce.

---

tributed environment: applications do not need to know whether the services these *Resources* provide are implemented locally or remotely, with respect to the application host and process. Because *Resources* themselves can be passed around as first-class network object parameters, Resources that implement local requests are transparently converted to handles (proxies, or stubs) for remote objects, when appropriate.

To intern an object, the `Listener` compares the nonce of all incoming client stubs (after the `RMarshaller` unmarshals the object encoding). If the incoming nonce matches any of the nonces for installed `RServices` on the `Entity`, the `Listener` simply passes on a handle to the `RService` instance, discarding the client stub.

The interning mechanism ensures that if *Resources* are passed back to their originally hosts, the real `RService` instance (rather than stubs) is properly passed onward.

3.2.2 Events

As mentioned in Sections 2.3.4 and 2.3.5, the O_2S `Event` framework enables `RServices` to send and receive asynchronous messages.

`Events` are simple *RType* structures similar to dictionaries that contain a variety of generic fields available for application-specific semantics. While these fields can contain any *RType*, the usage idiom is to strive for a lightweight `Event` payload. Table 3.3 lists these fields, along with suggested semantics.

Table 3.3: Event fields and their suggested semantics.

Field	Suggested Semantics
message_type	general event type
thrower	*Resource* which throws the event
recipient	event routing
message_string	human readable message
data	machine readable message
parameters	additional parameters

To send and receive Events, RServices rely on the Event Listener. As mentioned, the hosting Entity is designed to provide RServices with event services, so to obtain an Event Listener, the RService simply requests one from the hosting Entity. RService can request as many Event Listeners as necessary for the application with no additional overhead; as such, some applications may benefit in designating different Event Listeners for receiving different types of Events (or Events from different sources).

When RService request an Event Listener from the Entity, the RService also registers a callback method. When the Event Listener receives an incoming Event, the Event Listener calls the callback method (with the received Event as a parameter). The callback method is usually a method on the RService designated with the necessary logic to act upon incoming events.

The desired effect is that each Event Listener is "wired" with a target "address" (or callback), capable and designed to handle those Events. As such, handles to these Event Listener can then be passed to any remote host, thereby allowing multiple hosts to send Events to an RService.

To achieve this effect, Event Listeners are implemented as a special RService, with a special method named throw_event(). Event Listeners are then passed to remote hosts (where they become client stubs); when remote hosts wish to send Events, they call the throw_event() method, with the Event message as an argument. The Event Listener on the serving host receives the Event via the RService infrastructure and forwards the Event to the proper callback. Figure 3-6 illustrates this process.

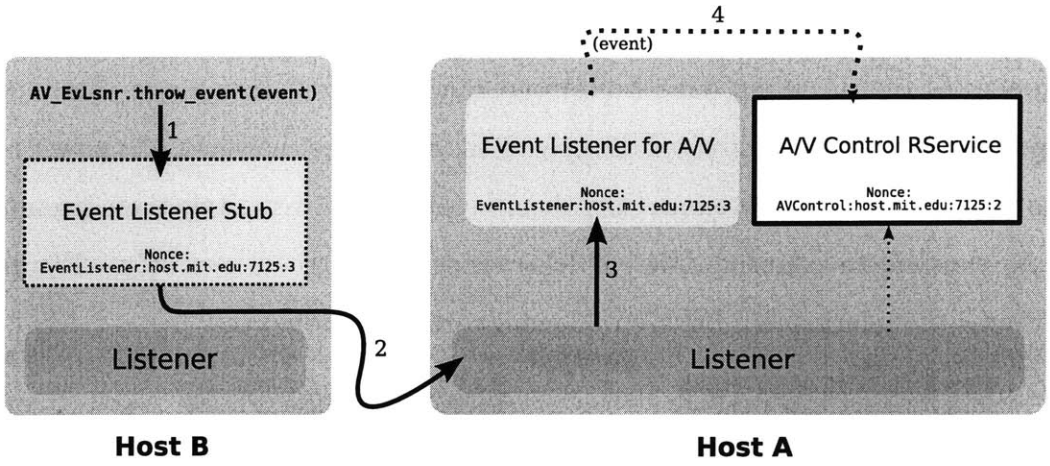

Figure 3-6: The `Event Listener`. `Events` from Host B are forwarded via the `Event Listener` stub to Host A, where the `Event` is passed to the `RService`'s callback.

3.2.3 Stream Connectors

Stream Connectors (`SConnectors`) are similar to `Events` in implementation design. The `SConnector` is a special `RService` with methods designed to open and control TCP/IP data streams. Similar to `Event Listeners`, `RServices` can obtain as many `SConnectors` as necessary via request to the Connector Manager in the `Entity`.

`SConnectors` are unidirectional; when requesting a `SConnector`, the `RService` must specify whether an input or output `SConnector` is desired. If the `RService` specifies an input `SConnector`, the `RService` also specifies a callback method, which is called every time the `SConnector` receives new incoming data. Conversely, the `RService` pushes data through an output `SConnector` by passing the data as an argument to the `SConnector`'s `send()` method.

When two appropriately gendered `SConnectors` are connected, they serve as a byte stream from the input `SConnector` to the output `SConnector`. The `SConnector` infrastructure also supports a sideband data stream encoded alongside the byte stream.

The external network API for controlling `SConnectors` is fairly limited: it allows remote hosts (with handles to these `SConnectors`) to simply connect `SConnectors` together, probe their state, and cut their connections. In the usage idiom for `SCon-`

nectors, the RService either provides the data source for output SConnectors — or processes the incoming data from input SConnectors. In a sense, the input or output of SConnectors are wired directly to the RService; however, the power of this architecture is that the SConnectors themselves can be wired together dynamically during runtime to connect different RServices together as necessary. Figure 3-7 illustrates the relationship between SConnectors and RServices.

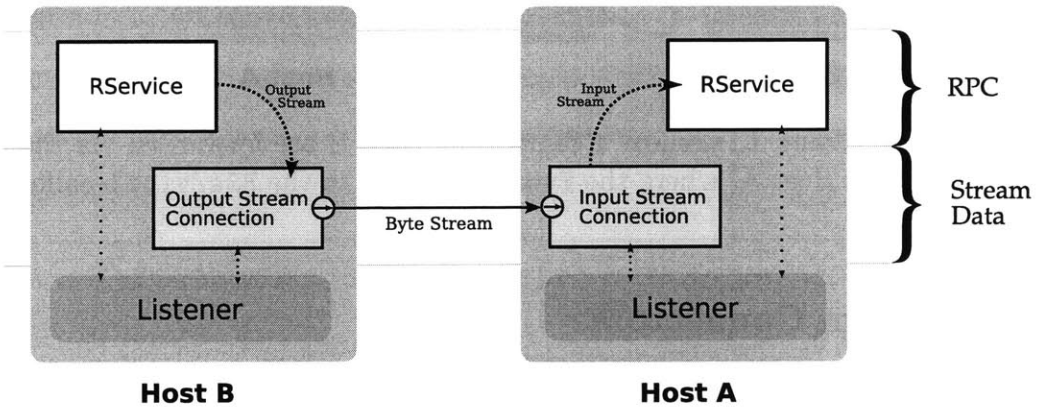

Figure 3-7: Unidirectional SConnectors are requested by RServices, which provide the data source for output SConnectors or data processing for input SConnectors. This also illustrates the system component layering described in Section 2.1: the RPC layer is used to control the faster, asynchronous byte stream layer.

As such, SConnector instances are usually requested from RServices by some application logic (often running on yet a different host). The application logic is generally cognizant of the functions provided by developer's RServices, so it is appropriate that the application logic selects the SConnectors from the desired RServices and connects them together to compose dynamic applications.

3.2.4 The Entity

The Entity is implemented as yet another special RService. The Entity instantiates one Listener (and then registers itself with its Listener) and provides well-defined methods for hosted RServices to access Event Listeners and Stream Connectors.

Since the Entity represents a logical host machine, the Entity is characterized by a variety of parameters that specify the name, location, and function. These

parameters can be set by the developer through environment variables or subclassing the generic Entity class.

In hosting and managing a collection of RServices, the Entity represents these RServices to the O_2S world at large. To publicize the Entity's existence, it registers with the Registry upon instantiation.

3.3 The O_2S Registry

The Registry provides a look-up service for finding other Entitys in the system; a keep-alive service, to monitor the liveliness of registered Entitys; and, a subscription-based notification service to alert interested parties when a certain Entity disconnects or registers. Registrys in principle broadcast the their host address on the local subnet, thereby enabling Entitys to discover the local Registry.[3]

3.3.1 Lookup Service

Entitys generally register with the Registry when they instantiate. The registration process entails registering the Entity's parameters with the Registry, as well as passing a handle of the registering Entity to the Registry. In doing so, others can search for registered Entitys, as well as obtain handles to them. The Registry enables Entitys to query and obtain handles to other registered Entitys, thus facilitating the bootstrapping process of obtaining the first remote Entity.

To search for Entitys, a typical query to the Registry usually specifies the desired values to the keywords that correspond to the identifying parameters of Entitys. A query can specify as few or as many keyword/value masks; the Registry returns a list of all Entitys (and their *Resource* handles) that match.

To support this search capability, the Registry implements a small internal database composed of several hash tables, keyed on the different parameters values, thereby ensuring a constant $O(1)$ lookup time (with respect to the number of registered Entitys at a given time) for each query. This implementation is highly

[3]In the current implementation, all Entitys are "wired" to use a common Registry.

advantageous, since the search and key/value matching functionality is used quite often for subscription notifications.

3.3.2 Subscriptions & Notifications

Often a `RService` will rely on the existence and liveliness of another `RService` on a different `Entity`; however, if the latter `Entity` is disconnected or fails, the former `RService` (or some other application logic) should expect an outage notification.

Since the `Registry` provides liveness monitoring, the `Registry` can also notify interested parties whenever any given `Entity` is disconnected. These notifications are implemented with O_2S `Events`, and interested parties must *subscribe* an "interest" with the `Registry` to receive liveness notifications for specified `Entitys`.

A party subscribes by first providing the `Registry` an `Event Listener` to receive notification. Additionally, the party must also specify which `Entitys` should trigger a notification to be sent; the party does this by providing a keyword/value mask of `Entity` parameters. The party also specifies whether a notification should be sent when the interested `Entity` registers, disconnects (fails), or both.

Whenever a new `Entity` registers or a registered `Entity` fails, the `Registry` compares this `Entity` against the list of subscriptions. Notification `Events` are sent to subscribers whose mask matches the new or failing `Entity`'s parameters.

3.3.3 Keep Alive

The `Registry` monitors `Entity` liveness with periodic UDP "pings". UDP is ideal for keep-alive monitoring, since UDP is considerably lightweight and avoids the overhead of using persistent TCP connections for the same purpose. The `Entity` is expected to send a UDP packet to the `Registry` periodically; in principle, the `Registry` determines the frequency that `Entitys` must send these UDP packets.[4] During registration, the `Registry` specifies to the `Entity` the frequency in which to send UDP packets.

[4]In the current implementation, this frequency is currently hard-coded at 2 seconds. The architecture, however, allows for future `Registry` versions to pick an appropriate frequency based on the `Entity`'s measured network latency.

The UDP packets nominally contain the Entity's nonce,[5] as well as a time-stamp. The encoded nonce enables the Registry to determine the Entity source of the UDP packets for the necessary bookkeeping.

If an Entity's keep-alive UDP packet fails to reach the Registry within the allotted interval window, the Registry first marks the Entity as a "Zombie" to indicate that the Entity's liveness state is currently in flux. The Registry waits several intervals to account for network latency, but after a configurable number of intervals, the Registry marks Entity "Dead," removes the Entity from the live database, and sends the appropriate notifications to other Entitys.

3.4 Garbage Collection

In terms of reference tracking, RServices are by default untracked: remote references to these RServices (client stubs) are not tracked by the system. While this is sufficient for most applications, sometimes applications may benefit from having access to the list of "client" Entitys (an Entity different from the tracked RService's hosting Entity) that possess handles to an RService.

If RServices need access to their client Entitys, these RServices must specify (during instantiation) that they should be tracked. Whenever a handle for a tracked RService is passed to a client Entity, the system automatically notifies the hosting Entity (of the tracked RService) with the identity of the client Entity. To achieve this, the encoding for tracked RServices also includes the nonce of the hosting Entity; hence, whenever tracked RServices are passed to another client Entity and unmarshalled, the system sends an Event to the hosting Entity encoded in the nonce.

The hosting Entity performs the necessary bookkeeping to map the each hosted tracked RService to client Entitys which possess handles on the tracked RService. It should be noted that the hosting Entity is always notified whenever *any* Entity in possession of a handle to the tracked RService passes the handle to some other

[5]Recall that Entitys are special RServices, and therefore the Entity itself has a nonce.

`Entity`.

Additionally, when tracked `RService` remote handles are garbage collected (by the language or interpreter) or explicitly terminated by the client `Entity`, the system also notifies the hosting `Entity`.

In turn, `Entitys` provide tracked `RServices` with an interface to obtain their tracking record. While tracked `RServices` can poll the `Entity` for a listing of client `Entitys` that possess a handle, the `Entity` also provides a callback feature: tracked `RServices` can register a callback method that the `Entity` calls whenever the location for any of the tracked `RService`'s handles change.

3.5 Implementation Technology

As mentioned, the O_2S architecture is language- and platform-independent to support a heterogeneous set of implementation technologies and devices typical of pervasive computing environments.

3.5.1 Language Independence

O_2S is currently implemented in both Python 2.3 and Java 1.4, as well as a prototype implementation in ANSI C. Developers can create and run `RServices` in either Python or Java; handles to `RServices` implemented in Java (as with all *RTypes*) can be passed to Python environments and vice versa.

3.5.2 Platform Independence

In principle, the O_2S architecture has no operating system specific dependencies. The system runs on nearly all major operating systems, since Python and Java are widely implemented; it has been tested on Linux 2.4, Windows, Mac OS X, as well as Familiar Linux [14] for iPAQ hand-held computers.

Additionally, there is a partial Java J2ME [15] implementation for the Nokia Series 60 mobile phone. Since the phone has limited network connectivity support,

this O_2S implementation is adapted to use Bluetooth for networking to a nearby O_2S-Bluetooth"proxy". The proxy both represents the mobile phone to the O_2S environment and communicates relevant messages back from the O_2S world to the phone.

Chapter 4

Evaluation and Applications

This chapter presents performance benchmarks for the *Resources* implementation described in Chapter 3, followed by descriptions of various application implementations that benefit by utilizing the *Resources* framework.

4.1 Benchmarks

These benchmarks measure both RPC and stream connector performance. For the RPC benchmarks, the first test measures the round-trip time for a null method call between two hosts; this benchmark attempts to measure the overhead of the system. The second benchmark measures the round-trip time for a collection of various *RTypes* that travel to one host and back (an "echo" request); this benchmark aims to measure the marshalling overhead. Finally, streaming data performance measures the overhead in transferring byte streams. Table 4.1 tabulates the results of RPC benchmarks; Table 4.2 presents the streaming data benchmarks.

In all tests, the designated "client host" is a Pentium 4/2.66GHz with 256MB RAM running Linux 2.4.27; the "server host" (running the `RService`) is a Pentium 3/1.13GHz with 512MB RAM running Linux 2.4.21. These machines both reside on the same 100MBit subnet and run Python 2.4 and Java 1.4.2.

Table 4.1: Benchmarks. Times are in milliseconds.

Client \ Server	null-method		data-echo	
	O_2S Python	O_2S Java	O_2S Python	O_2S Java
O_2S Python	8.18	5.77	87.43	43.28
O_2S Java	5.89	4.15	52.45	21.62
Sun Java RMI	0.97		10.52	

4.1.1 Null Method Resource Round-Trip Time

Table 4.1 shows the round-trip time for an empty-method call between a client and the `RService` server. The Python and Java implementations are procedurally identical.

These time values are the result of executing the empty-method call 100 times and averaging the result. For Java implementations, the Just-In-Time compiler begins to optimize performance after several iterations; hence, before the results are recorded, several burn-in iterations (10) are run and discarded for all benchmarks.

4.1.2 Resource Data-Marshalling Performance

In the data-marshalling test, the client constructs a dictionary containing a large variety of *RTypes* and passes this dictionary as an argument to a procedure on the `RService` server object. The `RService` object simply returns ("echos") the argument back to the client. This test involves the process of marshalling and unmarshalling a multitude of *RTypes* on both the `RService` and the client. The dictionary includes: a string, an integer, a double, a boolean, a list containing the above types, and a *Resource* handle to the server `RService`.

As with the empty method benchmark, the time values in Table 4.1 present the average of 100 runs, after 10 initial burn-in runs are discarded.

Table 4.2: Streaming Data Benchmarks.

	C TCP/IP Sockets	O_2S Stream Connectors (Python)	Java RMI
Time (ms)	96.100	179.38	854.80

4.1.3 Java RMI

For comparison, Table 4.1 also presents the performance of Java RMI [16] for both benchmarks under the same hardware platform and configuration. For the dictionary structure used in the data-marshalling benchmark, the RMI server stub is used in lieu of the `RService` handle. As with the tests above, the table values represent the average of 100 runs.

4.1.4 Streaming Data

These benchmarks suggest a five-fold performance cost for the `RService` (RPC-based) implementation when compared to native Sun Java RMI; this cost may be an artifact of the `RService` implementation still being in its early stages.

RPC, however, composes only part of the O_2S components architecture described in Section 2.1. While RPC is responsible for constructing, connecting, and controlling distributed modules, the majority of computation and data flow make use of stream based `SConnectors`, which introduce small overhead to the underlying operating system's TCP/IP implementation.

Table 4.2 shows the result of sending a 1MB file (filled with random bytes) in 1KB increments between two hosts (the same client and server machines used in the above tests). The 1MB file is sent in 1KB increments to simulate the "stream"-like process by which these modules would continually receive, process, and forward data to other modules. This benchmark compares the performance between standard C TCP/IP sockets, O_2S Stream Connectors, and Sun Java RMI. Java RMI performance is measured by invoking a remote method call to the server for each 1KB data segment. Table 4.2 reports the average of 10 runs for each platform. The O_2S implementation introduces some Python overhead compared to standard C sockets but still yields a 5-fold performance increase over Java RMI. Future O_2S implementations may benefit by implementing the streaming connector-level infrastructure in C to remove the Python overhead.

4.2 Applications

Another dimension for evaluation involves examining the ease of developing real-world, pervasive applications using the *Resources* framework. This section describes a few applications deployed in the O_2S system.

4.2.1 User Devices and Environment

The *Resources* framework facilitates building higher level constructs for managing users and devices. The O_2S system as a whole employs a variety of "proxies," implemented as `RServices`, to represent and interface the O_2S system with both user preferences and their devices. These proxies are generally layered hierarchically, as depicted in Figure 4-1 and further described below.

Host Proxies

Host Proxies represent computation resources or devices (e.g., hand-held computers, mobile phones, projectors, A/V systems, printers, and so forth). In general, Host Proxies are `RServices` that provide the interface between a physical device and the O_2S world, thereby representing these devices in the pervasive environment. Because Host Proxies implement the `RService` interface, they effectively present a coherent abstraction for developers to access and control a multitude of disparate devices, eliminating the need to address a myriad of APIs in the application logic.

User Proxies

Since pervasive environments aim to provide a human-centric computing experience, it seems natural to include proxies that represent users and their preferences. Currently, the O_2S User Proxy simply manages the user's collection of devices (*à la* Host Proxies) as depicted in Figure 4-1.

Just as Host Proxies provide an interface between the *Resources* abstraction and the physical device, User Proxies provide the next abstraction level. Since User Proxies (in principle) track user state and store user preferences, the User Proxy

serves as an interface between the O_2S world and the user's (Host Proxy) devices. By querying User Proxies, applications in the O_2S system can request access to a user's device; in return, the User Proxy can select the best device based on the user's state and preferences.

Room Proxies

Room Proxies are similar to User Proxies; instead of managing devices for people, Room Proxies manage devices attached to a particular room (e.g., displays, projectors, printers, and A/V systems). The Room Proxies provide an API for the O_2S system to query and access Host Proxies of room devices.

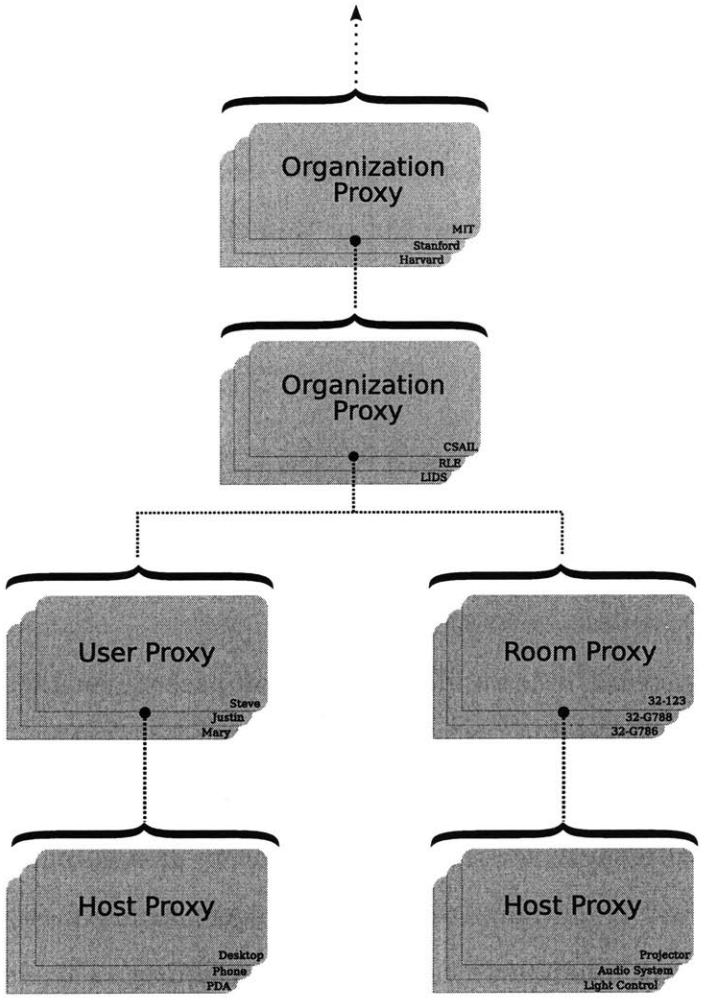

Figure 4-1: Proxy Hierarchy.

Organizational Proxies

While yet unimplemented, Organizational Proxies in principle manage a variety of Room Proxies, User Proxies, and especially other Organizational Proxies (in a hierarchal manner). For instance, an Organizational Proxy can take the form of a Laboratory Proxy, representing User Proxies for lab members and Room Proxies for lab seminar rooms. There may also be multiple Laboratory and Departmental Proxies under a University Proxy, and so forth. In principle, this hierarchal structure of proxies lends itself well for looking up any *Resource* in the system, in a similar fashion to DNS [17].

4.2.2 Heavyweight Computation For Lightweight Computers

Another class of applications that benefit from the *Resources* framework involve offloading computationally-intensive processing to dedicated servers. These servers can then provide their services to lightweight or thin clients, especially when it would be infeasible to run these services locally on the clients themselves.

Voice Recognition

One such application is voice recognition. A Voice Recognition server bundles a standard voice recognition package into a `RService`, which returns recognized tokens from utterances sent from remote, thin clients.

The Galaxy system [18] is an ideal recognition package for the O_2S Voice Recognition server implementation. Galaxy is suitable because the recognition engine supports multiple, concurrent recognition grammars and requires no voice training. Clients interact with the Voice Recognition server by first submitting a grammar to the server. The server initializes a Galaxy recognition instance with the specified grammar and returns a handle to the requested recognizer. The client then sends utterances via `SConnectors` to the recognizer; in turn, the recognizer returns recognized tokens back to the client via `Events`.

4.2.3 Distributed Applications

The notion of off-loading computationally-intensive processes to servers suggests that doing the same for application logic may also simplify distributed applications across multiple devices.

Data Hubs

O_2S Hubs provide the utility function of connecting multiple `SConnectors` together. Hubs provide an unlimited number of input and output `SConnectors`; data flowing through any of the input `SConnectors` is automatically forwarded to all output `SConnectors`. Applications generally utilize the Hub server to effortlessly connect multiple arbitrary components together, with the property that all components can broadcast data to each other.

The Hub resource illustrates the power of `SConnectors` in dynamic, pervasive environments. The Hub is a generic *Resource* that an application can employ to connect and disconnect arbitrary components during runtime. This feature enables applications to dynamically select optimal components and connect them in ways unanticipated by the developer.

Dynamic, Adaptive Chat

One such example application that makes use of the Hub server is the O_2S Adaptive Chat application, designed to facilitate a multi-user chat conference. The novel aspect of the chat application is that the application automatically finds the best device available for each chat participant (during runtime) and connects these devices together for a multi-modal conference.

The Chat application logic resides in the O_2S Planner (refer to Section 1.4.1). After the Planner performs the necessary planning, it makes use of the O_2S *Resources* framework to obtain handles to various resources, most notably users' devices and the Hub resource, and connects them together with `SConnectors`. The planning for this application generally involves contacting the User Proxies of all chat participants to

request access to the most feature rich, chat-appropriate devices (Host Proxies) owned by the user. Based on the user's detected location and set preferences, each User Proxy will return the most relevant devices to the application logic. The application then installs GUI resources (see below) on these devices and appropriately wires their respective input and output `SConnectors` to the Hub server.

The dynamic-nature of the Hub server provides the foundation for the adaptiveness of the Chat application. These chat instances are adaptive in that if any participant gains access to better devices (or their current devices fail) during the chat, the application logic automatically switches the user's device (by reconnecting `SConnectors` to the Hub) to gracefully upgrade or degrade the chat experience, all while maintaining the conversation.

Combined with the Voice Recognition server, the Chat Application enables users to dictate commands directly to the application logic. The application logic achieves this by requesting a Voice Recognition *Resource* and connecting it to an appropriate audio source on the user's device. Recognized tokens are sent (via `Events`) to the application for processing.

Graphical User Interfaces

Executing application logic on centralized servers instead of clients lends itself naturally to off-loading the logic for controlling graphical user interfaces (GUIs) as well. The Simple Unified GUI *Resource* (SUGR) is an interface that enables applications to separate the GUI presentation from the GUI logic. When the application logic resides on the server, the application is typically simplified by keeping the necessary GUI logic on the server as well, while disembodying the GUI presentation to the user's device.

At the heart of SUGR interface is the SUGR GUI-specification. The specification allows applications to specify both placement and the callback events for a variety of generic widgets. To present a GUI to on user's device, applications typically send a SUGR GUI-specification to the user's SUGR-enabled client. The SUGR-client then renders (using any GUI package) the GUI on the client device as specified in the

specification.

The SUGR interface also allows for applications to obtain *Resource* handles to the widgets for updating content during execution. The client receives all GUI events from the user and forwards them back to the application for processing.

The SUGR architecture is highly advantageous for developers, as it supports the natural tendency to keep application logic and GUI logic together.

4.2.4 Visualization

To facilitate debugging of distributed `RServices` and `SConnectors`, the *Resource* Visualizer (`RViz`) presents a real-time rendering of the *Resource* network graph, as shown in Figure 4-2[1].

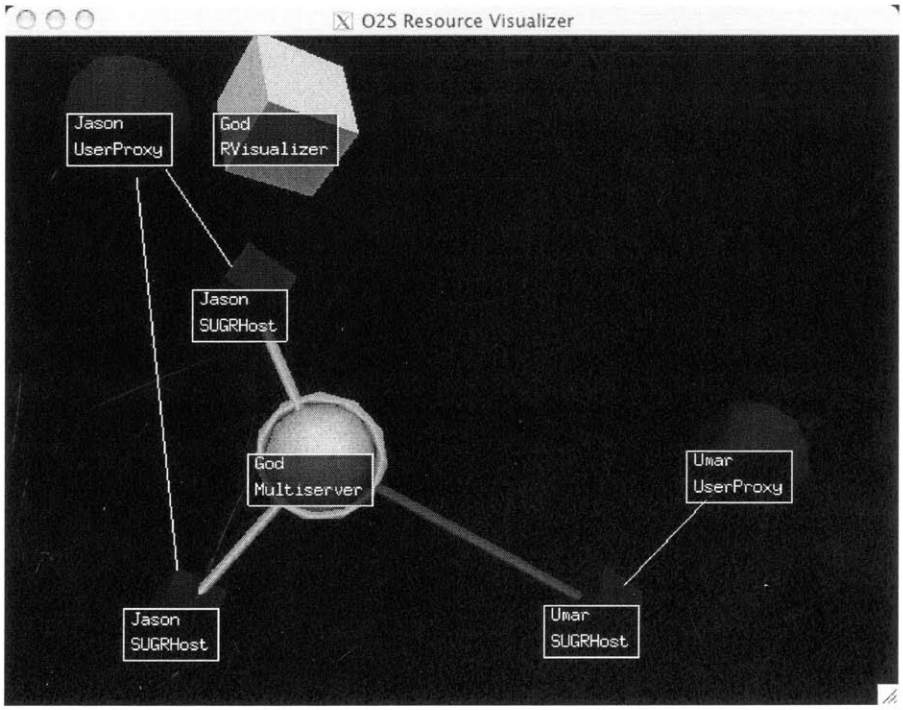

Figure 4-2: A typical `RViz` screen shot.

Nodes and edges represent `Entitys` (hosts) and the `SConnectors` between them, respectively. The `RViz` system is composed of an `RViz` server and `RViz` clients. The

[1]While there are several standard command-line and GUI-based implementations for diagnosing and inspecting *Resources*, `RViz` also presents `SConnectors` connections between the hosts.

RViz server, which communicates with the Registry to determine the state of the O_2S system, sends state updates to all RViz clients. The RViz clients render the state information, using Dot [19] for graph layout and OpenGL [20] for rendering. By employing the client/server model here, multiple RViz clients can render a view of the system concurrently.

4.2.5 Temptris

No software architecture is complete without a game. Temptris is a multi-player, distributed version of the popular falling-blocks game. In contrast to the Adaptive Chat application described above in Section 4.2.3, a considerable amount of the physics and animation logic for Temptris resides on the client side, as the user experience for action games is usually very dependent on high frame-rates and timely response to user input. As such, Temptris can run as a stand-alone single-player application (as most games do) but includes the capability to communicate with other players for a multi-player experience. Figure 4-3 is a screen shot of game play in Temptris.

Multiplayer Temptris relies on the Temptris Server (a RService), which runs on a dedicated server, to manage game instances between different players. The Temptris Server runs the multiplayer game logic, keeping track of the players, the current field height for each of the players, and establishes the order for the falling bricks. The latter feature ensures that all players receive the same blocks and in the same order. Furthermore, the server receives most state change events from game clients and can instruct clients to penalize players by adding additional lines to the board.

The achieved effect is that when any player clears n lines from their field, all other players (opponents) are penalized with $n - 1$ additional lines[2], which is added to each opponent's board to hasten her demise. This encourages players to play more aggressively, for the last player standing is the winner.

[2]When a player clears four lines at once ($n = 4$, e.g., a "*Tetris*"), all other players receive 4 lines.

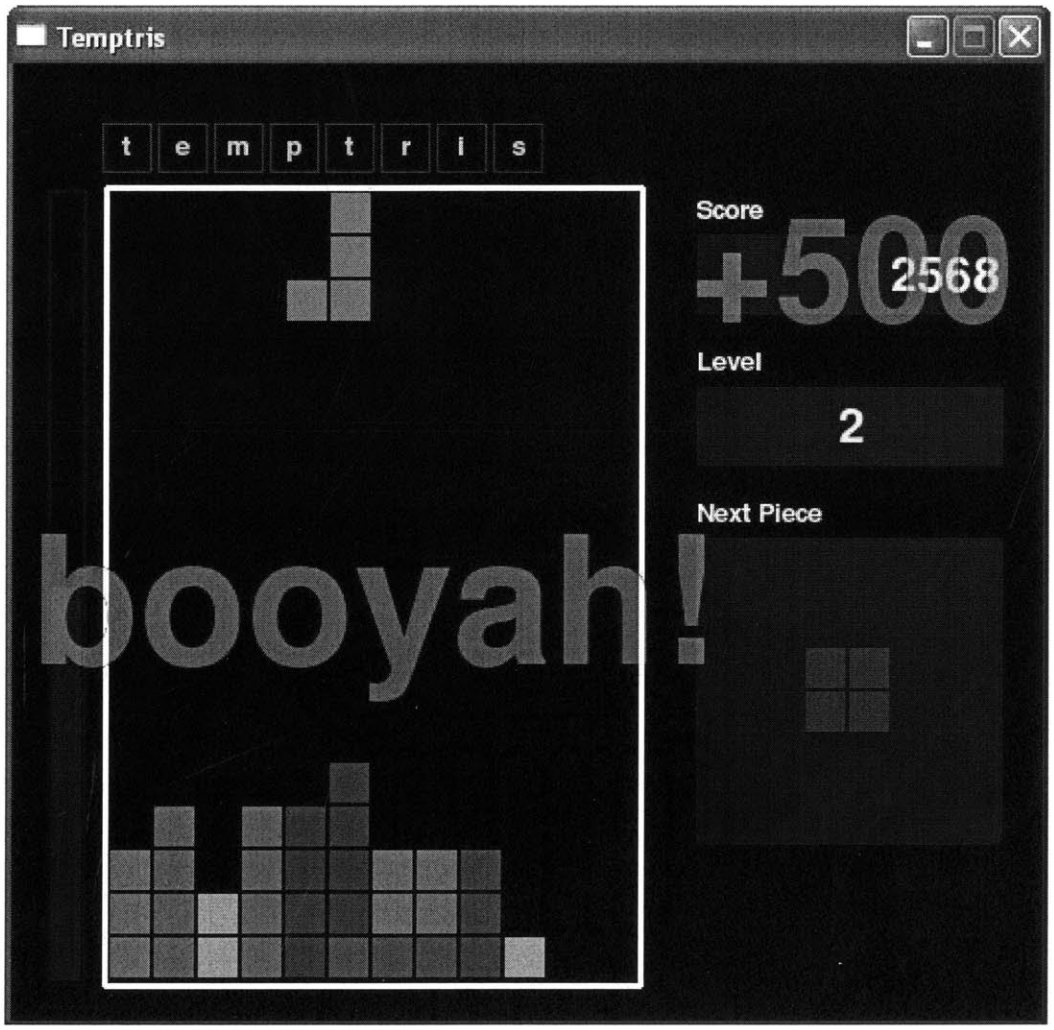

Figure 4-3: Temptris in action.

Chapter 5

Conclusion

This chapter discusses related work, future directions for the O_2S *Resources* framework, followed by concluding thoughts.

5.1 Related Work

Here are a few related RPC and component systems:

5.1.1 CORBA

The Common Object Request Broker Architecture (CORBA) [21] is a rich framework for developing distributed applications spanning different languages. At the heart of the CORBA framework is the Object Request Broker (ORB) Core, which handles all the necessary communication and data marshalling. In general, developers must specify the interfaces between the client and server at compile time using CORBA's Interface Definition Language (IDL); the interface is subsequently statically linked into the client and server applications.

5.1.2 Sun Java RMI and Jini

Sun's Remote Method Invocation (RMI) [16] augments the Java environment by providing a tightly integrated remote procedure call package. Java RMI's strengths

include an RPC semantic consistent with the Java programming idiom, a built-in security framework, a flexible object serialization framework, and optimized performance that capitalizes on JIT compilation.

However, RMI requires a fair amount of development effort. After developers define their remote object interfaces, they must also distribute copies of these interface class (bytecode) files to each client host requiring access to the remote object. Developers must also generate stubs and distribute them to all involved client hosts as well;[1] unfortunately, this mechanism forces developers to know *a priori* where (which hosts) services will run. Finally, developers who use RMI are limited to developing their network objects only in Java.

Jini

The Sun Jini Architecture [22] is service-oriented architecture designed for dynamic discovery and incorporation of network services. Jini is based upon RMI (and therefore requires Java) and enables devices to discover services on the local network via a Lookup service. Devices access services via RMI, but in the Jini framework, devices automatically download the necessary stub ("proxy") to these services. Jini relaxes some of RMI's requirements, namely those requiring the developer to know beforehand the location of remote services. Jini also features a leasing mechanism, where by clients and services negotiate resource allocation.

5.1.3 Metaglue

The Metaglue System [23] is a platform for developing distributed agents geared towards intelligent environments. The system extends the Java language, providing new constructs to program software agents. In the Metaglue system, distributed agents (or modules) can lookup and gain access to other agents and services. Agents specify their own environmental requirements for execution (such as dependencies on

[1]Recent versions of Java allow developers to place remote object stubs in a network-accessible location, such as a web-server, for clients to download on-the-fly. Unfortunately, this mechanism requires that users run a web-server, and in practice, Java's security framework makes for a very involved setup process.

other agents), and Metaglue attempts to accommodate these requirements either by instantiating the necessary dependency agents or by migrating agents appropriately.

5.1.4 Summary

Many of these systems subscribe to the conventional distributed application model, where applications are composed of statically-partitioned client-server modules which communicate using fixed, pre-defined APIs. In all of these systems, there is no distinction between mechanism and policy: the function implemented by a module and the policy specifying the role of the module in the overall logic of the application are inextricably intertwined in the code. With these traditional network object platforms, it is often difficult to adapt such distributed applications by replacing their constituent components, or to even reuse components for satisfying a different goal.

5.2 Future Work

While the current O_2S *Resources* implementation described in this work realizes many of the original ideals of the project, the following sections outlines several additional research directions to explore.

5.2.1 Composites & Hot-swapping

Composites are special *Resources*, which encapsulate one or more Composites (or *Resources*) and their interconnections. Composites serve as a convenient "black-box" container for a group of connected, constituent Resources, which collectively implement higher-level functionality.

Composites would facilitate the O_2S Planning Engine, which generally "wires" *Resources* together to implement a high level application. Composites then serve as another convenient abstraction layer (and handle) to a bundle of connected *Resources* that serve some coherent function. Figure 5-1 illustrates an example.

These Composites would feature a standardized API for constructing new Com-

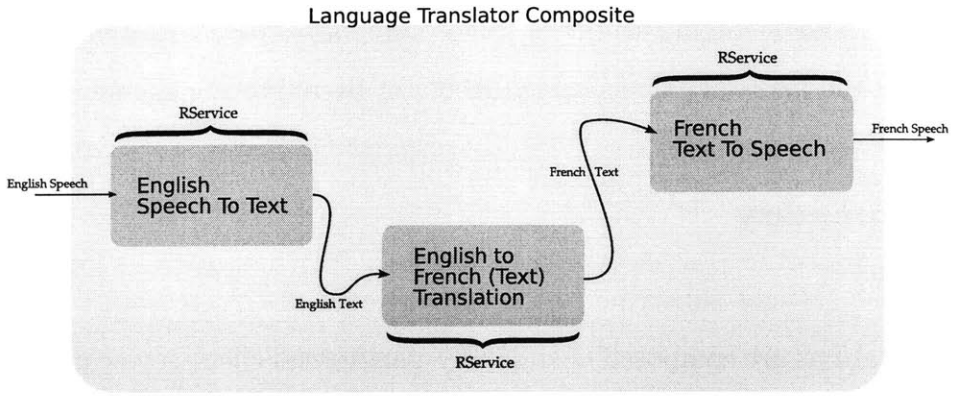

Figure 5-1: A Language Translation Composite, composed of multiple, connected `RServices` bundled together.

posites from existing Composites, by connecting the inputs and/or outputs of existing Composites to other Composites or *Resources*. In this way, developers can have handles to abstract each level of implementation.

The Composites abstraction lends itself nicely to "hot-swapping" features, in which the system can replace an entire Composite (or black-box functionality) with another congruent Composite instance – during runtime. Hot-swapping provides a natural abstraction for developing dynamic applications that adapt gracefully to user environments.

5.2.2 Remote Instantiation

With the O_2S *Resources* framework described, all real execution code resides on the server in the `RService` object instance and must be instantiated during system startup (or instantiated at a later time by a specialized `Entity`). However, the developer often cannot anticipate all the possible `RServices` that users may eventually require. Remote instantiation of `RServices` may solve this need by downloading and executing code modules during runtime. The (source or byte) code for these special `RServices`, or *Pebbles*, would be stored in a network-accessible repository and migrate during runtime to `Entitys` for execution upon request.

5.2.3 Security & Authentication

The current implementation of the O_2S *Resources* framework lacks any security, authentication, or access control mechanism. These are obviously important issues to address before the framework can be deployed outside the laboratory environment. However, given the flexible design of the O_2S *Resources* platform, augmenting the system with an authentication and security layer fits naturally within the architectural constructs. Future architects could implement these layers with additional servers or proxies, which might use off-the-shelf solutions, such as Kerberos [24], for authentication and key management. Incorporating signed and verifiable code-bases would also benefit the safety of *Pebbles* system for applications requiring remote invocation.

5.3 Conclusion

The O_2S system is a goal-oriented approach to satisfy user intent within pervasive computing environments. The system accomplishes this by promoting a strong separation between mechanism and policy. Maintaining this separation is difficult with traditional distributed applications; hence, a new programming abstraction is needed.

This thesis explores a programming abstraction and proposes an architecture characterized by a simple, synchronous environment for programming dynamic and adaptive applications. Developers easily construct new application functionality by connecting together a set of distributed, generic, and re-useable code-modules, much like wiring together a circuit of components. Once the application circuit implementation is constructed, the circuit runs autonomously, sending a serialized stream of events to the application logic for handling. Developing adaptive applications, which involves simple restructuring of generic code modules, becomes a more natural process.

The work finds that the abstraction does indeed facilitate building adaptive applications. With a simple, synchronous environment for connecting together components, distributed applications are easier to construct and debug; furthermore, incorporating run-time adaptiveness is simple and natural with this abstraction.

Providing clean abstractions for constructing adaptive applications entails a per-

formance cost: each RPC method call in the O_2S environment incurs a five-fold cost over Sun's Java RMI. However, by separating mechanism from policy, the performance cost only applies in the process of constructing the application circuit with modules. Once the implementation is constructed, what matters is the implementation's execution speed, which is based on the performance of the modules and their data stream connections. The benchmarks reveal that the implementation is indeed fast, as the module connections are comparable to raw network sockets. As a result, an abstraction that separates mechanism from policy provides a goal-oriented system with a simple environment to construct new application functionality from fast, highly parallel networks of interconnected modules.

Bibliography

[1] R. A. Brooks. The intelligent room project. In *CT '97: Proceedings of the 2nd International Conference on Cognitive Technology (CT '97)*, page 271, Washington, DC, USA, 1997. IEEE Computer Society.

[2] Andrew J. Wayne, Andrew A. Zucker, and Tracey Powell. So what about the "digital divide" in K-12 schools? In *The 30th Research Conference on Information, Communications, and Internet Policy*, 2002.

[3] Mark Weiser. The computer for the 21st century. *Scientific American*, 1991.

[4] MIT Computer Science and Artificial Intelligence Laboratory. Project Oxygen: Pervasive Human-Centered Computing. `http://oxygen.csail.mit.edu/`.

[5] Mark Weiser. The coming age of calm technology. Technical report, Xerox PARC, 1996.

[6] M. Satyanarayanan. Pervasive computing: Vision and challenges. In *IEEE Personal Communications*, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Umar Saif, Hubert Pham, Justin Mazzola Paluska, Jason Waterman, Chris Terman, and Steve Ward. A case for goal-oriented programming semantics. In *System Support for Ubiquitous Computing Workshop at the Fifth Annual Conference on Ubiquitous Computing (Ubicomp '03)*, 2003.

[8] Justin Mazzola Paluska. Automatic implementation generation for pervasive applications. Master's thesis, Massachusetts Institute of Technology, 2004.

[9] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, 1967.

[10] Bruce Jay Nelson. Remote procedure call. Technical report, Xerox PARC, 1981.

[11] International Organization for Standardization. Open systems interconnection – basic reference model: The basic model. ISO/IEC 7498-1:1994.

[12] XML-RPC Specification. `http://www.xmlrpc.com/spec`.

[13] C. Perkins. IP Mobility Support. RFC 2002 (Proposed Standard), October 1996. Obsoleted by RFC 3220, updated by RFC 2290.

[14] The Familiar Project. `http://familiar.handhelds.org/`.

[15] Java 2 Micro Edition. `http://java.sun.com/j2me/`.

[16] Sun Microsystems. Java Remote Method Invocation. `http://java.sun.com/rmi/`.

[17] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035.

[18] S Seneff, E Hurley, R Lau, C Pao, P Schmid, and V Zue. Galaxy-II: A reference architecture for conversational system development. In *ICSLP*, 1998.

[19] S.C. North and E. Koutsofios. Applications of graph visualization. In *Proceedings of Graphics Interface*, 1994.

[20] OpenGL Architecture Review Board, J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[21] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, 2.5 edition, September 2001.

[22] J Waldo. The Jini architecture for network-centric computing. In *Communications of the ACM*, 1999.

[23] M Coen, B Phillips, N Warshawsky, L Weisman, S Peters, and P Finin. Meeting the computational needs of intelligent environments: The metaglue system. In *Proceedings of MANSE*, 1999.

[24] Jennifer G. Steiner, Clifford Neuman, and Jeffery Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Technical Conference*, 1988.