REP: A Communication Mechanism for Pervasive Computing

Janet Davis

Department of Computer Science and Engineering

University of Washington

Seattle, WA 98195

jlnd@cs.washington.edu

June 5, 2001

Abstract

Pervasive computing provides an attractive vision for the future of computing. Computation will be everywhere, invisibly supporting users in their tasks. For this vision to become a reality, developers must create applications and services that adapt in a highly dynamic, ad-hoc, distributed system. In this paper, we explore the use of remote event passing (REP), an asynchronous message passing system for the *one.world* architecture, as an alternative to RPC. REP is intended to provide several benefits to developers of pervasive computing applications, including awareness of distribution, primitives for mobility transparency, a helpful but general communication model, and support for dynamic composition and interface evolution.

1 Introduction

In the vision of pervasive computing [29], computers will become so common and so unobtrusive that they blend into the fabric of everyday life, effectively becoming invisible. Computer users will focus on their tasks, rather than on the technologies they are using. To achieve this vision, a variety of devices will be deployed throughout our living and working spaces. Many devices will be mobile, and will be expected

to provide useful services with or without the support of a network. These devices will come together in a large, dynamic, distributed system without a single designer or administrator. With inexpensive CPUs becoming increasingly common in a range of devices, including palmtop computers, cell phones, cars, toys, and appliances as well as servers and workstations, computation is becoming truly ubiquitous. Yet, pervasive computing applications remain scarce. The challenge for developers is to build applications that adapt in a constantly changing heterogeneous environment.

Communication will be a significant aspect of many pervasive computing applications. Mobile applications will need to discover location-dependent services, such as printers and building maps. Applications running on impoverished devices such as palmtop computers will take advantage of computational resources available through the network. Networked sensors and appliances will cooperate with each other and with smarter devices. Of course, pervasive computing applications will also help people access shared information and communicate with each other. Although communication protocols for many services will be standardized, these protocols will continue to evolve over time and across different organizations.

Remote procedure call (RPC) [3] has long been the predominant model for communication between components of distributed systems. A remote procedure call looks like an ordinary procedure call to the programmer, but the call is executed by a different process, often on a different node. RPC's adherence to familiar programming models makes it easy to use and has contributed greatly to its popularity. However, this transparency may be counterproductive in a pervasive computing environment. Why is this? First, RPC seeks to hide distribution through the use of programmatic interfaces. Not only do many implementations hide network, node, and service failures, but application developers may view such failures as a rare occurrence. With pervasive computing, the failure or unavailability of some resource will be the rule rather than the exception. Different applications will require different failure recovery and adaptation strategies, so programmers must expect and account for failures. Second, in order to provide this programmatic interface to remote resources, RPC uses stubs that fix the service interface at compile time. This limits opportunities for application developers to allow for interface evolution by negotiating an interface at run time. Moreover, RPC hinders dynamic composition, because it is tedious to interpose on a wide programmatic interface. Finally, RPC is only one point in the space of possible communication models. Although the client-server

model and the assumption of one response for each request has worked well in many distributed systems, pervasive applications may benefit from other styles of communication that fit poorly with our notion of a procedure call.

As an alternative to remote procedure call, we developed remote event passing (REP), an asynchronous, typed message passing system. Although event passing is less familiar and less well developed than threading as a means to structure applications, we believe it provides several benefits. REP makes communication explicit, so that programmers are aware of distribution and are prepared to adapt to a changing environment. At the same time, the simplicity and directness of the REP interface allows us to integrate service discovery and late binding [1], which can be used for mobility transparency. The event passing model is natural in applications that must react to real world events, but is general enough to allow other styles of interaction, including request/response interactions similar to RPC. Furthermore, REP supports dynamic composition and interface evolution by communicating through passive, semi-structured data rather than through programmatic interfaces.

The remainder of this paper is structured as follows. In Section 2, we discuss related work in communication systems for distributed computing. In Section 3, we describe the design of REP in more detail, along with its implementation in *one.world*, a software architecture for pervasive computing [8]. In Section 4, we present an evaluation of the performance and effectiveness of REP. We conclude and discuss future work in Section 5.

2 Related work

Although the idea of RPC had been around for several years [31], Birrell and Nelson's implementation in the early 1980s [3] was the first transparent RPC system. In this implementation, a call-based interface is provided by client and server stubs compiled from a service interface specification. These stubs draw on a runtime system to manage all the details of communication. Remote procedure call typically provides the semantics of an ordinary procedure call: communication is synchronous, and exactly one value is returned to the calling process. Such calls appear identical to ordinary procedure calls as long as there are no com-

munication failures; similarly, providing a remotely accessible service is nearly as simple as providing a local run-time library. Birrell and Nelson's stated goal was to make distributed computation easy. Commercial RPC implementations, such as Sun RPC [15], were quick to arise, and, as Birrell and Nelson had hoped, RPC soon became the dominant communication paradigm for distributed computing. Although message passing is more flexible, remote procedure calls are easier to understand and reason about [21, 14]. The semantics of a remote procedure call are familiar to all programmers.

Several efforts have explored the modification of RPC semantics to include asynchrony, either for concurrency or for disconnected operation. Notably, promises [14] are a linguistic mechanism for asynchronous remote procedure call: although a remote call is started when a promise is created, the calling process does not block until it attempts to claim the value of the promise. This allows communication and call execution to occur in parallel with the calling program. The Rover system [11] uses Queued RPC to allow communication in a mobile computing environment, where network connectivity is intermittent and may be variable in quality and cost. Remote procedure calls are queued while nodes are disconnected, and they may be reordered so that high priority calls are performed first when network connections are slow or expensive. Also, in some situations, it may be necessary or desirable for the return path to use a different protocol than the original invocation. Although pervasive applications must deal with the same network connectivity issues, REP takes a fundamentally different approach. Rather than masking the inability to contact a service, we expose it, to allow the application to react. The application will have its own strategy for adapting, which may be to queue the event until connectivity returns, but it may be to choose a different service or do without the service entirely.

Although RPC semantics are often desirable, sometimes they are not. For instance, several requests may be answered with a single response, or the response may come from a different process than the original recipient of the request [13, 28]. RPC is ill-suited to group communication, and to situations in which there is no clear client-server relationship such as Unix pipelines [25]. We can expect similar situations in pervasive computing services and applications.

RPC is certainly not the only communication model used in distributed systems. The space of possible models has been widely explored. In 1982, Spector published a taxonomy of communication systems,

parameterized along a number of dimensions including synchrony, reliability, the presence or absence of a return value, and whether the communication can be handled by a kernel process or must be handled by a user process [22]. Arguments and return values may be untyped or typed, and typed arguments may be passed by reference, by value [9], or by move [12]. Node and network failures may be masked or exposed.

We briefly discuss systems at several points in that space. The communication mechanism provided by the V distributed system [4] is similar to RPC in that it is synchronous and requires a return value, but it uses a message passing metaphor. Active Messages [27] are an asynchronous communication mechanism in which each message contains the address of a handler to be executed. It is similar to RPC in that the message causes a specific block of code to be executed, but the role of the handler is to get the message off the network rather than to perform computation and produce a result. VMTP [5] is a transactional transport protocol; requests are paired with responses, and a client can have at most one request outstanding. It is intended to support conversational protocols such as RPC, but also supports group communications, datagram delivery, and request forwarding. The Horus system [26] provides point-to-point and group communications through a message passing interface. By providing a standard protocol interface for protocol composition and a number of basic building blocks such as checksumming, retransmission, and service discovery, Horus supports a wide variety of communication models. GTS [16] provides composable protocols for group communications similar to Horus, but accounts for network partitions by using a server to queue messages for disconnected group members.

The oldest of these, and the most similar to REP, is CLU's strongly typed, asynchronous message passing primitive [13]. At the time that Liskov argued for this primitive, distributed computing was a new field, as pervasive computing is now. Therefore, Liskov chose a flexible, low-level primitive, rather than a high-level primitive that might have precluded desirable solutions. REP is similar to CLU in that it implements an asynchronous message passing model and does not mask node or network failures. However, where REP provides a uniform interface for communicating semi-structured data, CLU requires that message ports declare the types of messages they accept and send, similar to an RPC interface. Although this allows compile-time type checking, it precludes dynamic interface negotiation. Furthermore, unlike CLU, REP provides transparency to mobility through late binding and service discovery.

3 Design and implementation

REP is an asynchronous, message passing system with a simple, uniform interface. It provides a single interface for both point-to-point communication and service discovery and supports both early and late binding. In the remainder of this section, we explain the reasoning behind our design, describe the API provided for remote event passing in *one.world*, and discuss some of the details of its implementation as a *one.world* service.

3.1 Design

Our goals in designing REP were to increase programmer awareness of distribution in a changing environment while providing useful primitives for adapting to change, to provide a useful and general communication model, and to support dynamic composition and interface evolution. We now discuss how each of those goals influenced our design.

REP exposes communication rather than hiding it behind a programmatic interface. The programmer must explicitly send a message to a remote resource, rather than invoking a stub method. This ensures that programmers are thinking about communication where it occurs. Network and node failures, as well as resource unavailability, are exposed through exceptional conditions, allowing application-specific responses to such failures. In the spirit of exposing failures, messages are delivered at most once. Moreover, by eschewing stubs, we avoid distributed garbage collection, a challenging problem for distributed object systems [20]. Remotely accessible resources' lifetimes are controlled by the programs that must supply memory and computation for them, rather than by other programs that hold references to them.

Eliminating interface and failure transparency makes it easier to provide another kind of transparency: mobility transparency. We are able to provide service discovery with late binding using the same API as for point-to-point communications. Like the Intentional Naming Service [1], the REP interface provides expressive, machine-independent naming based on records of attribute-value pairs. Such names provide a level of indirection allowing an application to seamlessly continue communicating with services even as nodes and services move through the network or as service attributes change. Late binding integrates message

routing with name resolution, increasing responsiveness to change. We also provide early binding, where a resource name is explicitly resolved to a reference to a specific resource before messages are sent to that resource. A reference obtained through early binding is guaranteed to refer to a particular instance of a resource and will fail when the resource does. Early binding is necessary when the communication end-points share session state, because unlike late binding, it guarantees that a sequence of messages will be sent between the same end-points. The failure of the reference notifies an end-point when the other end-point is gone and a new resource must be bound.

Event passing is a useful and natural metaphor when dealing with real-world occurrences. Furthermore, events are used in large-scale services for high throughput, scalability, and load conditioning [7, 19, 30] and in small devices such as network sensors for ease of design and power efficiency [10, 17]. It is reasonable to provide an event-driven interface to these devices and services. At the same time, asynchronous message passing is a general enough communication model to support the development of other protocols, as observed by Liskov [13]. For instance, timeouts and retries can be used to implement at-least-once delivery. Message passing can be used in request/response interactions, asynchronous or synchronous, and even longer conversations. Resources may be leased across node boundaries at the application level. Although these protocols might be implemented more efficiently at a lower level, experimentation is required to determine which protocols are useful enough for pervasive computing to justify such an implementation.

Finally, we use a simple, uniform interface for handling passive, semi-structured events. The uniform interface simplifies interposition, because there is only a single point at which to interpose rather than a wide, programmatic interface. Dynamic composition is simplified due to the absence of compile-time type constraints. The exchange of semi-structured data allows events to be extended with optional data for optional functionality. Even if the communication end-points do not agree about protocol extensions, the presence of fields that are unknown to one end-point or the other should not interfere with their ability to communicate using the basic protocol.

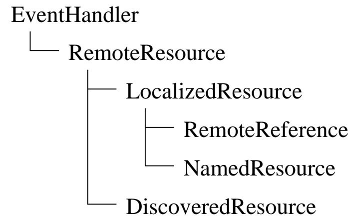

Figure 1: The `RemoteResource` class hierarchy.

3.2 API

The remote event passing API depends heavily on the conventions of *one.world*, a system architecture for pervasive computing implemented in Java. In *one.world*, data is represented by *tuples*, semi-structured records of typed attributes, while functionality is implemented by *components*. Control flow between components is expressed not by procedure calls, but by passing *events*. An event is a kind of tuple with two special fields:

- the *source*, an event handler that will receive any exceptional conditions related to the event and that will receive the response in a request/response interaction, and
- the *closure*, an arbitrary object that the application may use to maintain state related to the event and to help disambiguate the responses and exceptions it may receive.

Thus, in addition to providing communication through events, the interface to REP itself is based on events.

Significant aspects of the API include naming remote resources, exporting event handlers as remote resources, sending and receiving remote events, and explicitly resolving resource names. We discuss each of these in turn.

With REP, it is possible to name specific event handler instances, instances of a named resource on a particular host, and resources that are described by a tuple through the discovery service. As shown in Figure 1, remote event handlers are named with instances of the abstract `RemoteResource` class. The `RemoteResource` class does not handle events, but it descends from the `EventHandler` class so that they may fill event slots that are reserved for event handlers, such as the event source. The `LocalizedResource` class specifies a host name and a port number, localizing the resource to a particular host. A `RemoteReference`

```
EventHandler requestHandler;
Object closure;
RemoteResource destination;
RemoteReference myReference;
EventHandler lease;

...

public void handle(Event e) {
  if (e instanceof EnvironmentEvent
      && (EnvironmentEvent.ACTIVATED == ((EnvironmentEvent)e).type)) {

    // Export myself anonymously
    RemoteDescriptor descriptor = new RemoteDescriptor(this);
    BindingRequest br = new BindingRequest(main, null, descriptor, 0);
    requestHandler.handle(br);

  } else if (e instanceof BindingResponse) {
    BindingResponse response = (BindingResponse)e;
    myReference = response.resource;
    lease = response.lease;

    // Send a remote event
    requestHandler.handle(new RemoteEvent(this, closure, destination,
                                      new RequestEvent(myReference, closure));

  } else if (e instanceof RemoteEvent) {
    if (e instanceof RemoteEvent) {
      // Got a response for my remote event
      Event responseEvent = ((RemoteEvent)e).event;

    } else if (e instanceof ExceptionalEvent) {
      Throwable x = ((ExceptionalEvent)e).x;
      if (x instanceof ConnectionFailedException) {
        // No connection to remote node
      } else if (x instanceof UnknownResourceException) {
        // No resource matches the description
      }
    }
  }
}
```

Figure 2: Event handler code for a simple request/response interaction. When the application is activated, it exports itself as a remote resource. It then sends a request and awaits a response.

has a globally unique identifier (GUID). A new GUID is generated each time an event handler is exported; they are never reused. Thus, a `RemoteReference` is used to name a specific instance of an event handler. A `NamedResource` has a string-based name in addition to a location, and refers to a service on a particular host. A `DiscoveredResource` names a resource to be located by the discovery service. It contains a boolean query over the attributes and values of the resource descriptors, as well as a flag indicating whether anycast or multicast is desired.

We illustrate the remainder of the REP interface with an example of a simple request/response interaction, shown in Figure 2. In this example, the `requestHandler` is an event handler that will accept requests made

to the REP service, the `closure` is an arbitrary object that the application event handler will use to detect responses to its request, and the `destination` is the name of some service on a different node.

The *export* operation allows an event handler to be named so that it may receive remote events. Event handlers are exported using *one.world*'s binding mechanism, specifically the `BindingRequest` shown. When remotely exporting an event handler, the binding is between an event handler and its names. The `BindingRequest` must contain a descriptor for the resource to be bound. In this case, we wish to export the event handler anonymously, so the `RemoteDescriptor` contains only a reference to the event handler itself. The `BindingRequest` is sent to an event handler for the REP service, and the application event handler gets a `BindingResponse` in response. The resource contained in the `BindingResponse` is a `RemoteReference` that refers uniquely to this event handler. The `BindingResponse` also includes a lease that controls the binding between the event handler and the `RemoteReference`. This allows the exporter of the event handler to have explicit control over the duration of the binding, while allowing the binding to expire if the exporter crashes or loses interest.

The event handler can also be exported with a node-specific name or to the discovery service using the *export* operation. In the example, the only modification required is in the contents of the `RemoteDescriptor`. To export the event handler with a node-specific name, the `RemoteDescriptor` constructor would be called with a string as its second argument, in addition to the reference to the event handler itself. The event handler could then be referred to using a `NamedResource` as well as a `RemoteReference`. If the second argument to the `RemoteDescriptor` constructor is an arbitrary tuple, this tuple will be used as the event handler's name, and the binding will be propagated to the local discovery service.

Next, we consider the *send* operation. This operation, implemented by `RemoteEvent`, allows an event to be sent to the event handler named by a `RemoteResource`. The `RemoteEvent` constructor has four fields: the event source, the closure, the destination, and a nested event. The nested event may be any type of event, defined in the *one.world* core or by an application or service. In our example, the nested event is a `RequestEvent`. The remote reference for the event handler is given as the source of the nested event so that it may receive responses to the event. The nested event must contain only symbolic event handlers, because our goal is to exchange passive data rather than functionality. If the destination of the remote event is a

localized resource, the event will be sent directly to the named host; otherwise, it is routed by the discovery service.

If the service named by `destination` successfully receives the `RequestEvent` and sends a response, the event handler will receive a `RemoteEvent` containing the service's response. REP delivers the `RemoteEvent` itself rather than only the nested event so that the event handler can be aware it is receiving events from a remote source. The event handler can invoke itself on the nested event, if desired. If the REP service is unable to contact the node that is hosting the destination service, the event handler will receive an `ExceptionalEvent` containing a `ConnectionFailedException`. An `UnknownResourceException` signals that the named resource could not be found.

The final operation, *resolve*, allows a `NamedResource` or a `DiscoveredResource` to be explicitly resolved to one or more `RemoteReferences`. This operation is necessary only when using early binding to engage in a stateful communication protocol. It is implemented by the `ResolutionEvent` class.

3.3 Implementation details

REP is a centralized service running in the *one.world* kernel. This makes using REP convenient for *one.world* application developers, because there is no need to explicitly link against a REP component. Also, services exported via REP are accessible via a single, well-known port. However, this means that one instance of the REP server is shared by all services and applications running on the *one.world* node. If isolation is desirable—for instance, when developers want to run both release and debugging versions of a service—a REP component may be instantiated as part of an application.

The *one.world* implementation of REP takes advantage of *one.world*'s structured I/O facilities. The structured I/O interface, which is used for both storage and communications, permits a number of operations of applications over tuples, including *put* to write a tuple and *listen* to receive all tuples matching a specified query as they are written. Tuples are marshaled using Java serialization. The current REP implementation communicates over structured I/O channels that use TCP as their underlying transport. Although this adds a layer of indirection to REP communication and contributes to event passing latency, it lets us leverage our work on structured I/O and thereby simplify the REP implementation. Moreover, building on top of the

structured I/O interface should allow us to use protocols other than TCP, such as UDP, HTTP, or email.

REP binds structured I/O channels on demand and caches them for later use. Caching allows one channel to be shared across multiple operations, amortizing the cost of channel establishment and reducing latency when multiple events are sent within a short span of time. Events are queued while a connection is being established. The channel is stored in a hash table based on the remote node's IP address and port number. It is removed from the cache when it is unused for some period of time specified in the *one.world* configuration file. The current default is one minute; real applications should be studied to learn a more appropriate timeout. Channels are also removed from the cache when the connection has dropped, due to network failure or the channel being closed by the remote node. If an attempt to send an event on a cached channel fails because the connection was dropped, an attempt is made to resend the event in case connectivity has been recovered.

4 Evaluation

In this section, we present an evaluation of the performance and effectiveness of REP. Our primary point of comparison is remote method invocation (RMI) [23], the RPC-like system that is included in Sun's Java software development kit. First, we characterize REP's basic performance through a range of microbenchmarks. Then, we evaluate REP's effectiveness and usability, drawing on a case study of students using *one.world* to build real applications as well as our own experiences. Our results show that REP has reasonable performance and support our hypotheses regarding transparency and appropriateness of the event-passing model. Further evaluation of interface evolution is required. Although some usability problems were discovered, we believe these can be overcome.

4.1 Performance

To determine the basic performance of remote messaging, we compare the latency and throughput of REP with that of RMI. All measurements were performed using off-the-shelf PCs with Pentium III 800 MHz processors and 256 MB of RAM, running Windows 2000. The PCs are connected by a 100 Mb switched

Benchmark	RMI	REP
Empty call	0.3	
Simple event	3.0	3.8
Large event	3.4	4.2
Complex event	3.8	5.5

Table 1: Remote messaging latency, in milliseconds.

Benchmark	RMI	REP
Empty call	4329	
Event without return	833	881
Event with return	609	751

Table 2: Remote messaging throughput, in calls or events per second.

Ethernet. Reported results are the average of 100 consecutive benchmark runs.

For latency, we compare the time required for request/response interactions using REP with that required for the same interaction using RMI. The results of four tests are presented in Table 1. In the first test, RMI is used to invoke a method with no arguments and no return value. In the remainder of the tests, an event is sent to the server, where it is immediately returned to the client. The events include a very simple event, a more complex event, and a large but simply structured event containing a 1000-byte array. The size of the complex event when serialized is slightly smaller than that of the 1000-byte array. A REP request/response interaction with a simple event is substantially slower than an empty call via RMI. However, when identical interactions are compared to take into account the cost of serializing and transmitting a larger amount of data, REP suffers only a 25% performance penalty in comparison to RMI. Using REP to send an event with a complex structure is significantly more expensive than sending a byte array of comparable size, supporting our conjecture that serialization is an important contributor to latency.

For throughput, we look at the number of events an REP server can receive in one second versus the number of remote method invocations a RMI server can process in one second. The results of three tests

are reported in Table 2. In the first test, an RMI server receives empty method calls. In the second test, the method that is invoked via RMI takes a simple event as an argument and returns null. The REP server does not send a return event. In the third test, the server echoes the event back to the client. As with latency, we find that REP performance is poor in comparison to empty remote method invocations, but the two systems are roughly comparable when both transmit the same data. In fact, REP throughput when passing events is slightly better than RMI throughput, whether there is a return value or not. We speculate that this is because the *one.world* core is multithreaded, allowing overlap of I/O operations with computation, whereas the RMI `UnicastRemoteObject` appears to be single-threaded.

These results show that remote messaging has reasonable performance, given the high cost of Java serialization. The *one.world* team is considering alternatives to reduce the overhead of serialization, including the use of pre-serialized objects [30] and dynamic code generation to specialize serialization.

4.2 Effectiveness and usability

To evaluate *one.world*'s effectiveness in building applications, we conducted an experiment in the form of a senior-level undergraduate project course. After ten introductory lectures, the nine students split into two teams that developed a music sharing system and a universal inbox. Each team split into two subteams; one used existing Java-based technologies, including RMI and Jini [24], while the other used *one.world*. Because both subteams implemented the same application, this experiment allows us to compare our architecture with existing approaches to building distributed systems. The results reported here are based on end-of-term interviews and the teams' final reports. We supplement these results with our own experiences in building a data replication system based on the two-tier replication scheme [6]. We focus on four areas: transparency, appropriateness of the model, composition and evolution, and general usability.

4.2.1 Transparency

Although the students found it easy to build a distributed system using Jini, making that system robust was not so easy. Due to the interface and failure transparency provided by RMI and Jini, only a few lines of code are required to transform an existing Java system into a Jini-based distributed system. Servers must

be exported, and clients must bind to them. However, making the system robust to network or service failures is a tedious process of iteratively testing the system, identifying failure cases, and modifying the system to account for those cases. The original system was not designed or written with such failures in mind. By contrast, the students using *one.world* considered distribution from the beginning in structuring their applications. Although they still had the often difficult task of deciding how best to react to failures, the students were aware when failures could occur. We had a similar experience in our development of the replication system. Awareness of distribution and a thoughtful accounting of failure conditions should result in more robust applications.

Furthermore, *one.world* provides a useful kind of transparency that RMI and Jini do not: transparency to mobility. As the students realized, discovery and late binding make it possible to move (or restart) services with minimal disruption to the system as a whole. The delivery of exceptions through exceptional events makes it simple to retry an attempt to contact a service after a brief loss of connectivity. There was no need for an explicit rebinding of the service.

4.2.2 Appropriateness of the event-passing model

The students found the concept of asynchronous event passing easy to grasp. Both applications used events as a significant form of communication, both in the *one.world* and the Java implementations. Event passing was a good fit for notification about real world events, as well as for callbacks and simple request/response interactions. Furthermore, our own experiences and previous work on scalable servers [19, 30], suggest that events are also natural for managing concurrency in pipelined or dataflow services.

Asynchrony was crucial for the students when dealing with untrusted services, which will be the rule rather than the exception in pervasive computing environments. The students assumed that services were not malicious but could be buggy. Jini provides an event passing system, but it is implemented using synchronous RMI calls, thus exposing applications to potentially indeterminate delays due to buggy services. The students using Jini used a `TaskManager` object—essentially, an explicit event queue—to divorce the processing of an event from its communication. Even this is not a perfect solution, since services must be trusted to use a `TaskManager` to ensure asynchrony. Moreover, although the `TaskManager` class is included in the Jini

distribution, it is neither documented in the distribution nor part of the Jini standard. With REP, events are truly asynchronous; no additional work was required to ensure isolation from unreliable services.

As for the generality of the REP model, the students were indeed able to use REP to implement application-level protocols with different semantics. Using REP and timers, the universal inbox team implemented a form of reliable event delivery in which every event results in a response indicating whether the remote resource was able to handle the event. Both we and the students implemented a number of other request/response interactions, with varying reliability. The music sharing application required an event forwarding system, which was simple to implement with REP. The students were able to maintain a stack representing the request's path in its closure, allowing the response to be propagated backwards along the path without maintaining any state on the nodes along the path. Alternatively, any responses could easily be sent directly to the request source, short-circuiting the path taken by the request itself.

4.2.3 Composition and interface evolution

Students appreciated the uniform event handling interface. They were able to send existing application events to services on remote nodes by simply wrapping the events in `RemoteEvents`. By contrast, the students using Jini and RMI spent a substantial amount of time learning to use the stub compiler, where to put the stub classes, and how to use the command-line options when running clients and servers. Interestingly, the Jini users implemented RMI proxies for their services rather than implementing the services directly as remote objects to allow for the use of different communication mechanisms. However, they chose a very narrow interface that was easy to interpose on.

Unfortunately, our experiment did not address the issue of interface evolution. An experiment that would address this issue by forcing divergent evolution in the short term is as follows. The experiment would begin with an implementation of a basic event-driven service and a client that used that service. At least two teams of developers would extend the service interface with new features, particularly features that require adding new data to existing event types, while maintaining the basic functionality. The developers would also implement clients that use the new features. The new clients and servers should then be tested against each other and against the basic client and server. If clients and servers are able to recover gracefully from

their disagreements about the interface and successfully communicate using the basic interface, then the experiment is a success.

Of course, more evidence regarding system evolution should be gathered by observing the evolution of applications over time.

4.2.4 Usability

Both we and the students found that asynchronous events are easy to use in services that provide a response to a request and where control flow is simple. As observed by Ousterhout [18], asynchronous events allow multitasking without true CPU concurrency. This simplifies programming by removing the need for locks and other mechanisms for concurrency management. Debugging is simpler in that execution order does not depend on thread scheduling. Moreover, we and students were able to encapsulate state in events and in short-lived event handlers rather than in shared application components.

However, debugging in *one.world* is not yet well developed. In our experience and that of the students, debugging in *one.world* is done primarily by logging important events and state transitions. *one.world* has a configuration flag that causes it to log significant kernel events. Applications can also be instrumented to print out event information as the events are processed. This is often tedious. More importantly, such logs give little information about causality: All that is certain is that one event was processed before or after another. The developer debugging the application does not know which of the previous events caused some particular event to be sent. We plan to take advantage of *one.world*'s uniform interface to allow the interposition of a debugger on application event handlers. To address the problem of causality, events could be annotated with the events that logically preceded them where appropriate.

Students found it difficult to manage asynchrony. Although we found that managing asynchrony was simple when control flow was simple, as with pipelines and request/response interactions, we too had difficulty managing asynchrony when the control flow was more complicated and involved loops or interesting failure recovery schemes. In these circumstances, state machines become too complex to easily reason about or to code efficiently and readably. For instance, while developing the replication service, we designed an eighteen-state machine for the task of binding and comparing the contents of two tuple stores, with error recovery.

Although we believe the state machine is correct, we were not able to implement it to our satisfaction.

We believe this difficulty in structuring asynchronous applications is due to a lack of design patterns for asynchrony. We explored one pattern in particular, the logic/operation pattern. In this pattern, asynchronous *operations*, parameterized by failure recovery strategy, are connected by blocks of synchronous *logic*, which do not fail except under catastrophic conditions. Logic and operations can be combined to form an *asynchronous function*, which is itself an operation, supporting structured programming. The logic/operation pattern should help to provide structure to components with complex control flow and simplify failure management. We plan to refine this pattern and use it to rebuild our replication service in order to evaluate its usefulness. Ultimately, we believe the best solution may be language support for asynchrony, including continuations and syntactic aids for specifying failure handlers.

5 Conclusion

In this paper, we have argued that the asynchronous exchange of semi-structured messages is a more suitable communication mechanism than remote procedure call for pervasive computing environments. Exposing distribution through event passing encourages developers to consider distribution from the beginning and build robust applications. At the same time, discovery and late binding provide transparency to mobility. Asynchronous events are a natural match when reacting to real world events and are helpful in dealing with untrusted services, but also can be used to build higher-level communication protocols. Although we have not yet tested our hypothesis that REP's semi-structured events help with interface evolution, we did learn that the uniform event handling interface simplifies development and application configuration.

Although asynchrony provides new challenges in structuring and debugging code, we believe REP remains a promising communication model for pervasive computing. To help overcome these challenges, we will develop debugging tools and design patterns for asynchronous event passing, including the logic/operation pattern. We will continue our evaluation by using these tools and patterns to re-implement the data replication service and develop other applications that will help us to explore mobility and communication. We also plan further work in performance optimizations and virtualization.

Although it is not the currently targeted use scenario, REP may be used for communication between components within the same *one.world* node, just as RPC is often used for communication between processes on a single host [2]. The REP service may be optimized for local communication by avoiding the overhead of event serialization and of passing events through the network stack. However, there is some cost associated with detecting events that are destined for the local node. Further study should determine whether remote or local communication is the common case, and whether the benefit of detecting local communication outweighs the cost.

We also wish to explore the possibility of REP using other implementations of the structured I/O interface, communicating via UDP, HTTP, or email. As with Queued RPC [11], virtualization would allow communication via a channel whose semantics are consistent with those required by the application or situation. UDP would provide less reliable but more efficient communication, HTTP can be used to cross firewalls, and email would allow communication when the communication end-points are connected to the network at different times. Virtualization will require changes to both the REP interface and implementation, as naming in point-to-point communication currently uses a host and port number. An alternative is to use the same naming scheme as is used for the structured I/O channels themselves.

Acknowledgements

Thanks to the other members of the *one.world* team, without whom this work would not have been possible: Robert Grimm, Eric Lemar, Adam MacBeth, and Steve Swanson. Thanks also to my advisors, Tom Anderson and Steve Gribble, for their patience and sound advice.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *Proceedings of the Twelfth Symposium on Operating Systems Principles, Operating Systems Review special issue*, 23(5):102–113, 1989.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] D. Cheriton. The V distributed system, March 1988.

[5] D. R. Cheriton. VMTP: A transport protocol for the next generation of communications systems. In *SIGCOMM '86: Proceedings of the ACM SIGCOMM Conference on Communications Architecture & Protocols*, pages 406–415, Stowe, VT, August 1986.

[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[7] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The ninja architecture for robust internet-scale systems and services, 2000.

[8] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. Systems directions for pervasive computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau, Germany, 2001.

[9] M. P. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, 1982.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, November 2000.

[11] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and F. M. Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 156–171, Copper Mountain, CO, 1995.

[12] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[13] B. Liskov. Primitives for distributed computing. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–42, 1979.

[14] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 260–267, 1988.

[15] B. Lyon. Sun remote procedure call specification. Technical report, Sun Microsystems, Inc., 1984.

[16] S. Maffeis, W. Bischofberger, and K.-U. Mätzel. A generic multicast transport service to support disconnected operation. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, Ann Arbor, MI, 1995.

[17] H. Muller and C. Randell. An event-driven sensor architecture for low power wearables. In *ICSE 2000, Workshop on Software Engineering for Wearable and Pervasive Computing*, pages 39–41. ACM/IEEE, 2000.

[18] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Annual Technical Conference, January 1996.

[19] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable webserver. In *Proceedings of the USENIX Annual Technical Conference*, pages 106–119, Monterey, CA, 1999.

[20] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, pages 211–249, Kinross, Scotland (UK), 1995.

[21] R. Schlichting and F. Schneider. Understanding and using asynchronous message passing. In *Proceedings of the ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, pages 141–147, Ottawa, Canada, August 1982.

[22] A. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4):246–260, 1982.

[23] Sun Microsystems, Inc. Java remote method invocation (RMI). *http://java.sun.com/products/jdk/rmi/index.html*.

[24] Sun Microsystems, Inc. Jini network technology. *http://www.sun.com/jini/*.

[25] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[26] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A framework for protocol composition in horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, 1995.

[27] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[28] R. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, pages 71–78, 1993.

[29] M. Weiser. The computer for the 21st century. *Scientific American*, September 1991.

[30] M. Welsh and D. Culler. Achieving robust, scalable cluster I/O in Java. In *Proceedings of the Fifth ACM SIGPLAN Workshop on Languages, Compilers, and Runtime Environments for Scalable Computers (LCR2K)*, Rochester, NY, June 2000.

[31] J. E. White. A high-level framework fror network-based resource sharing. In *Proceedings of the National Computer Conference*, June 1976.