A Framework for Active Distributed Services

Ramesh Govindan  
Cengiz Alaettinoglu  
Deborah Estrin

Computer Science Department and  
Information Sciences Institute  
University of Southern California  
Los Angeles, CA 90089  
Email: govindan,cengiz,estrin@usc.edu

January 1997

Abstract

Dynamically evolving a widely distributed service or application is a challenging systems design task. In this paper, we describe a plausible, high-level design for a common software framework that supports such extensible distributed services. Our design requires that the distributed service be implemented in an interpreted language, but our framework allows transparent and on-demand procedure instantiation at agents of the distributed service. In addition, this framework enables distributed services to be composed from reusable modules, and service sessions to be initiated from a single site.

1 Introduction

With the advent of a world-wide networking infrastructure, we increasingly see large-scale deployments of distributed services and applications. Sessions of some distributed services comprise tens or hundreds of cooperating agents (instances of the service executing at network nodes). Inter-agent communication is usually achieved using network-layer multicast or configured agent hierarchies. For example, the Harvest low-latency document retrieval service employs a static hierarchy of cooperating servers [4]. On the other hand, a popular Internet conferencing application, w6b [10], uses reliable multicast to ensure synchronized group state among conference participants. We are currently developing a decentralized Internet routing registry system in which agents use multicast for rendezvous and synchronization [1].

In future internetworks, we can expect increased deployments of distributed services and applications: services that disseminate time-sensitive information globally (e.g., news and stock quotes), distributed services that electronically broker world-wide transactions, and applications that enable multi-party scenario simulations (e.g., disaster response). Before such distributed services become ubiquitous, new mechanisms that simplify the development, debugging, deployment and evolution of these services are needed.

This paper discusses dynamically extensible distributed services. Specifically, we focus on enabling in-session extensions to distributed services. Such dynamic extensibility is particularly relevant to distributed services. These are feature-rich software subsystems with frequent need for functionality enhancements. But, much human coordination is necessary to achieve these enhancements in large-scale deployments; disruptions resulting from such service upgrades can adversely affect long-running sessions of highly-available distributed services.

To enable service extensibility, we leverage the trend towards using interpreted languages (such as Java [11], Safe-TCL [3], or Python [16]) for distributed services and applications [13]. Interpreters for such languages already support—or can be extended to support—dynamic procedure instantiation: that is, code for a procedure can be installed in the run-time environment after a program has started executing, but when, or before, the procedure is referenced. We can use this feature to support active distributed services, which are extensible at the granularity of individual procedures or modules. Mechanically, we envisage active distributed services using this feature in one of two ways: agents may lazily retrieve extensions upon reference, or one or more agents may eagerly distribute extensions to all other participants.

In this paper, we argue that the protocols and software subsystems needed to support such active distributed services are largely independent of the dis-
distributed service itself (Section 2). We then describe a plausible, high-level design of a framework for supporting active distributed services (Sections 3 and 4). Such a framework can also allow centralized initiation of a distributed service, and dynamic composition of distributed services from re-usable modules. Finally, Section 5 discusses related work.

2 Active Distributed Services

Why do distributed services need to be extensible? What kinds of extensibility can we anticipate of these services? The answers to these questions motivate the need for a common substrate for active distributed services (ADSs).

Motivation

Developers frequently enhance the performance and functionality of distributed services and applications. To upgrade all agents of a large distributed service session, significant coordination is required between the participants. If distributed services were explicitly designed to be extensible, they could avoid service disruptions caused by functionality enhancements.

More importantly, there exist foreseeable problem domains where ADSs are functionally necessary:

Broker Systems Future internets will introduce the ability to conduct worldwide commercial transactions. The core of this system will be a global collection of extensible brokers that mediate transactions between the participating entities. The brokers will cooperate to provide a set of common functionality required for scalable global transactions. Over time, these highly-available servers will need to be dynamically extended to accommodate new transaction types or methods.

Graphical Visualization In future internets, it will be possible to conduct wide-area, multi-party scenario simulations, e.g., for responses to natural disasters [5]. In these simulations, participants together visualize the output of models describing a natural phenomenon. This output usually consists of a composite, continuously changing, representation of several graphical objects. Due to the widely varying nature of these phenomena, it may be desirable to dynamically alter models or their parameters during the session. With an extensible graphical visualization application, an agent can introduce new graphical objects, or modify the behavior of existing graphical objects, in response to changes in models or model parameters.

Structure

Understanding the structure of ADSs helps us define their extensibility requirements. But, first, what is a distributed service? A distributed service consists of two or more agents each executing at different nodes of an internetwork. Several concurrent sessions of the distributed service may co-exist in the internetwork. In particular, a single node may contain more than one agent of the same distributed service, each agent belonging to a different session. All agents in a session execute the same, consistent version of code. Each agent communicates with at least one other agent using an application-specific mechanism.

What is the structure of an ADS? Every extensible system contains a mechanism for seamlessly incorporating extensions. In most extensible systems, an event-based invocation mechanism is used [21]. In this mechanism, extensions are usually designed as event handlers. Any component of the system may raise an event upon the occurrence of some system activity, at which time the corresponding extension is processed. The binding between an event and its handler can be dynamically altered, or delayed until the event is raised.

The granularity of events in extensible systems varies with the degree of extensibility permitted in the system. Extensible editing and mail handling tools pre-define hooks to which users can add customized processing code. Some recent work on extensible operating systems has proposed events at procedure granularity—every procedure call is an event, and the binding of the call to the called code can be changed dynamically [15].

Event-based invocation is, then, a reasonable model for describing the structure of ADSs. The granularity of events depends on the ADS and its desired degree of extensibility. In our visualization example, the developer can choose to allow users to describe not only the sequence of graphical operations on objects (e.g., at time T move object to x, at T + 1 rotate object by y degrees), but also modify the implementation of individual graphical operations like rotations, shears etc. Roughly speaking, for each extensible aspect of the system, the ADS developer would define one event.

Stated in terms of this model of extensible system structure, there are three ways in which ADSs may be extended:

- An agent may install a new event at all other agents. This corresponds to adding a new object to our visualization application. Not all ADSs will allow this level of extensibility.
- An agent may push an event handler implementation to all other agents. In our visualization example, the agent that alters the model parameters
might transmit new descriptions of object behavior to other handlers.

- In some sessions, agents may start and terminate at different times, so pushing handlers alone may not be sufficient. An agent may retrieve the handler implementation when the event is raised. This might also be useful in lazily retrieving infrequently used extensions.

Support for dynamic retrieval of handlers can also enable:

**Composability** To allow rapid prototyping of ADSs, it may be desirable to facilitate the flexible re-use of distributed service components. This can be particularly useful for the hard-to-debug inter-agent communication mechanisms. If these externally implemented modules are designed as extensions to the ADS, they can then be referenced upon demand.

**Rapid Initiation** For some distributed services and applications, it may be desirable to centrally initiate a network-wide session of the service. With dynamic handler retrieval, this is conceptually straightforward if an ADS defines every procedure to be an event. An initiator need only distribute the main procedure to participants; other procedures can be retrieved upon demand.

**Requirements**

From our model of the structure of ADSs, there emerges the need for three kinds of protocols. First, those that allow one agent to instantiate new events and event handlers at all other agents participating in the distributed session. Second, those that loosely synchronize the modification of event handlers at agents to avoid inconsistent event handling. Third, those that retrieve handlers on demand upon reference, or that allow pre-fetching of handlers. A primary concern in the design of these protocols is scalability to large, widespread sessions.

Our description of the kinds of extensibility required of distributed services is largely independent of a specific service or application. We argue, therefore, that ADSs are best served by implementing these protocols in a common run-time *substrate*. To enhance the usability of this substrate, and to thereby spur the development of ADSs, we impose three design constraints:

- the substrate’s mechanisms that support extensions to distributed services must, to the extent possible, be transparent to developers.

- The substrate must not require that all extensions to a service be implemented in the same language as the distributed service itself. This is useful for re-using modules implemented in a different language, for example.

- Finally, the substrate must not require modifications to language interpreters. This will allow the substrate to be used with commercially implemented or proprietary platforms.

The substrate is one component of a software framework that supports active distributed services. A second component is a *toolkit* that contains parametrizable implementations of re-usable communication modules. The collection of toolkit modules will evolve over time. However, to maximize the viability of this framework, it is crucial to have a widely usable initial collection of mechanisms in the toolkit.

3 The Substrate

From these requirements, we arrive at the following two-component design for the substrate (Figure 1):

**Actuator** At each node in the network, there is one actuator. The actuator initiates the execution of an ADS agent at the node. It also implements protocols that retrieve handlers on-demand, or push handlers to other agents in the session. Finally, it facilitates the invocation of handlers implemented in a different language. Before the actuator initiates an ADS agent, it may preprocess the agent code in a language-specific way. This step avoids modifying interpreters for dynamic handler retrieval.

**Envoy** A node may contain several agents belonging to different ADS sessions. Associated with each agent is an envoy. An envoy detects missing handlers and interacts with the node’s actuator to retrieve them. It also installs pushed handlers in the agent’s run-time environment. The functionality in the envoy is language and interpreter-dependent.

Why this division of functionality? The complex mechanisms for distributing extensions need only be implemented once (in the actuator), regardless of the number of different languages used to implement ADSs. Furthermore, these mechanisms are hidden from ADS agents, who interact with the actuator through their envos. Finally, envos contain only *glue* functionality, and can be easily ported to new languages.

We now describe these components in greater detail.
3.1 The Actuator

To instantiate an agent at a node, the actuator activates the corresponding language interpreter and envoy, and sets the interpreter to execute the ADS. The actuator also implements protocols for pushing event handlers to all participants of a session. When the actuator receives pushed event handlers, it installs them in the local agent. Conversely, when an envoy needs to retrieve handlers on demand, it contacts the actuator; remote procedure calls (RPCs) are used for all actuator-envoy communication. The actuator then retrieves the absent handler. To accommodate ADS-specific handler retrieval, the initiator of an ADS session may push a *retrieval handler* for the session. The actuator uses this to fault in missing handlers.

Support for ADS extensions implemented in a different language is a critical piece of the substrate functionality. How does the actuator enable cross-language handler invocation? Suppose, an actuator retrieves a handler on behalf of an ADS session. Suppose, further, that that handler is implemented in a language different from that of the ADS. The actuator first initiates a new interpreter and a new envoy for this language. It then communicates, to the two envosy, a generic interface description (using, for example, IDL [19]) for the retrieved handler. The envos generate the appropriate RPC stubs to complete the handler invocation.

Before the actuator initiates an ADS agent, it preprocesses the agent code. This step, which we call *priming*, customizes the ADS to fit the capabilities of the corresponding language interpreter. Specifically, if the ADS's language interpreter does not provide event handler fault notification, the actuator may need to pre-process the ADS to emulate that exception. The actuator must also prime retrieved or pushed handlers before handing them off to the respective envoy.

3.2 The Envoy

One envoy is co-located with each agent executing at a node (Figure 1). The process of priming an ADS or its extensions results in the envoy being invoked at run-time. The envoy performs three primary functions: it interacts with the language interpreter, it communicates with the actuator to retrieve missing handler, and it facilitates cross-language handler invocation.

To enable handler retrieval, an envoy must first be notified when the corresponding event is raised. This notification is language and interpreter specific. Some interpreters may throw an exception upon reference to an undefined handler—the envoy’s exception handler will initiate handler retrieval. In the absence of such interpreter support, the process of priming the ADS will emulate this exception.

The envoy then calls its local actuator, which dynamically retrieves the missing handler and hands it off to the envoy. The envoy installs the primed handler in the agent’s run-time environment, allowing the handler invocation to complete. For handlers implemented in a different language, the actuator passes to the envoy a formal procedural interface description instead. Using this, the envoy generates, *on the fly*, the caller stub for the RPC. Concurrently, the actuator activates the appropriate interpreter and envoy for the handler. The handler’s envoy generates the callee stub for the RPC, allowing the invocation to complete.
3.3 Some Research Directions

A key research issue for the substrate is the design of protocols for distributing ADS extensions. In the design of the actuator, we have allowed for ADS-specific extension retrieval methods. Nevertheless, for increased viability, the substrate will need to implement protocols that perform well over a range of session characteristics: sparse and dense participant distributions, localized or widely dispersed agents, and varying session membership dynamics.

But, what kinds of protocols are required for dynamically distributing ADS extensions? First, to retrieve a handler upon reference, actuators can search, in expanding multicast scope, for nearby actuators with copies of a handler. Second, for eager distribution of event handlers, the actuator will need to implement a reliable multicast protocol. The alternative, a low-rate periodic multicast of extension handlers, can add significant latency to handler execution. The reliable multicast protocol will be different from others in use today, since eager code distribution will involve a small number of sources and several hundreds or thousands of receivers. Third, the actuator will need a protocol to loosely synchronize instantiation of handlers in ADS agents. Why? In an ADS, it may be possible to replace an existing event handler with a new one. When pushing new handlers, different agents in a session can process the same application message using different handlers. This can lead to unpredictable ADS execution. Finally, the actuator must also implement protocols to ensure that all agents execute consistent versions of handlers. Especially in long-running sessions where not all agents are initiated at the same time, different agents in the same session may retrieve different versions of the handler.

Earlier distributed systems research has studied similar group communication and consensus protocols extensively. Why can we not use this work directly in implementing our inter-actuator protocols? Especially for widely dispersed groups, internetworks exhibit unpredictable delays, varying bandwidths, and network dynamics that temporally affect group membership. For this reason, distributed systems approaches that rely on frequent and tight synchronization between group members perform poorly over internetworks. Such approaches also scale poorly to large group sizes.

An approach that promises better performance and scale exploits application-specific characteristics, where possible, to more loosely coupled group members. This approach builds into the application the resilience to incomplete or outdated group state. It also relies on probabilistic control message transmission with suppression, and on low-rate advertisements that allow members to eventually reconstruct group state. We will use these techniques to adapt earlier distributed systems algorithms to the inter-actuator communication mechanisms described above.

Even so, to get acceptable performance for ADSs with many extensions, these group communication and synchronization protocols must be designed carefully. To illustrate this, we consider the performance impact of dynamic handler retrieval. To understand the quantitative impact of this aspect of our design, we conducted simple experiments to compare the time to service a page fault against multicast request/response latencies. The latter metric is, to a first approximation, the time to “fault” in a handler in the substrate. Figure 2 shows the interval between sending out a single multicast request, and receiving responses from hosts at four different multicast scopes: site, region, continent, and world.

From Figure 2, we see that handler faults serviced by participants “near” the requestor are within twice page fault service times. These numbers are encouraging. If our handler retrieval protocols allow any participant—not just the initiator—to respond to handler fault requests, retrieval overhead can be kept within acceptable limits for large sessions. For sparse, widely distributed sessions, the likelihood of finding a “nearby” participant is small. In these situations, retrieval latency can be reduced by prefetching, or eagerly distributing handlers. Partial procedure call graphs—generated, for example, when priming an extension (Section 3.2)—can be used to determine an optimal prefetching strategy.

Designing the substrate itself to be extensible represents a second research issue. At least two components of the substrate will evolve over time. First, some of the substrate components are language-specific: interpreters, envoys, and the code for priming handlers. As new languages appear, or as interpreter functionality evolves, we need mechanisms to distribute new versions of these components. For this, it probably suffices to designate some specific archival nodes in the internetwork. Then, rather than be configured with the identity of these sites, actuators can discover their existence using network-layer multicast. Second, the retrieval and synchronization protocols will themselves evolve with experience from deployments. A separate distribution mechanism should be used for these protocol extensions from that used for ADS extensions. Because every node in the internetwork will have an actuator executing at all times, a low-rate periodic flooding protocol might suffice for actuator extensions.

In our experiments, the requestor was at USC/ISI, and the responders were respectively located at USC in Los Angeles; Xerox PARC in Palo Alto, CA; Merit Networks Inc., in Michigan; and RIPE NCC in Amsterdam, the Netherlands.
### Table

<table>
<thead>
<tr>
<th>Metric</th>
<th>Time (in milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Fault Service Time</td>
<td>18</td>
</tr>
<tr>
<td>Site-Region Request/Response</td>
<td>5</td>
</tr>
<tr>
<td>Region-Continent Request/Response</td>
<td>35</td>
</tr>
<tr>
<td>Continent-World Request/Response</td>
<td>90</td>
</tr>
<tr>
<td>World-Request/Response</td>
<td>260</td>
</tr>
</tbody>
</table>

Figure 2: **Overhead of Dynamic Handler Retrieval**: An approximate measure of the overhead of dynamic handler retrieval is the latency of multicast request and response. If the responder is “near” the requestor, this overhead is twice the page fault service time.

Ensuring safe and secure ADS extensions is a third research issue. As with other extensible systems, ADSs are susceptible to improper extension behavior. Unsafe dynamic extensions can interfere, sometimes subtly, with existing system components. Unlike some other extensible systems ADSs may not need sophisticated firewalling against unsafe extensions. They will, however, need a scalable tracing and trace collection mechanism for after-the-fact analysis of the effect of unsafe extensions. Fine-grain execution tracing can be performed transparently by the substrate, either by exploiting interpreter capabilities, or by appropriately priming extensions.

Once efficient retrieval mechanisms have been designed, the performance of local actuator mechanisms will be a fourth research issue. Among these mechanisms, actuator-envoy communication, cross-language invocation, and extension priming are candidates for optimization, especially for highly extensible distributed services and applications.

### 4 The Toolkit

With delayed procedure instantiation, a developer can more easily compose an ADS from parametrizable modular implementations (which we call bundles) of communication mechanisms. For example, suppose that an ADS developer implements a bundle containing a reliable multicast transport protocol. Other ADS developers can call the entry points of this bundle; the code corresponding to the entry point will be retrieved upon reference. For example, in Figure 3(c), procedure foo() can reference a ReliableMulticast::send() procedure implemented by another developer.

We call the collection of reusable bundles the ADS toolkit. Three constraints affect the design of bundles and the toolkit. First, the substrate needs to support cross-language procedure invocation (Section 3) so that ADSs may invoke bundles implemented in a different language. It is unrealistic to expect that bundles will be implemented in all possible active programming languages. It is equally unrealistic to expect that there exists a single programming language for ADSs, or a single virtual machine for which all languages may be compiled. Second, so that distributed services may tailor bundles to their needs, the toolkit must contain parametrizable modules. This also avoids a proliferation of ad hoc implementations of substantially similar bundles. Finally, to maximize its viability, the toolkit must satisfy the needs of most distributed services.

While the last requirement is difficult to satisfy without experience from widespread deployments of ADSs, we can anticipate some common needs of distributed services. Examples of such needs include:

**Reliable Group Communication** Future distributed services will be built upon reliable wide-area group communication. With increasing availability of network-layer multicast, reliable multicast transport protocols can be used to realize such group communication. The design of such protocols is influenced by application and session characteristics such as size, locality of group membership, number of senders, and the message delivery semantics associated with the application. While no single reliable multicast protocol may satisfy the needs of applications, there may exist classes of parametrizable protocols which may be implemented as bundles.

**Self-Configuring Hierarchies** For scalable agent communication, many distributed services rely on agent hierarchies. Parent-child relationships in today’s agent hierarchies are usually manually configured into participating agents. Inter-agent communication is constrained by these relationships; an agent obtains information either directly from
Figure 3: An Active Distributed Service: It is desirable to initiate an ADS session by sending a single bootstrap segment containing the main() procedure (a). Subsequently, other procedures are “faulted” in (b), including procedures from other modules which were used to compose the ADS. (c)

its parent, or relies on its parents to recursively locate the required information [4]. Manually configured hierarchies can be inefficient, can be intolerant of failures, can require operator intervention, and may be susceptible to configuration errors. The toolkit can contain protocols for self-configuring hierarchies, parametrizable by application specific metrics.

5 Related Work

Several areas of work impact the design of a framework for active distributed services:

Programming Mobile Agents: Several other languages (e.g., Phantom [7], Obliq [6], and Telescript [23]) are designed for programming mobile agents. A mobile agent carries a transient execution environment along with code; this allows a computation to start at one node in the network, migrate itself to another node and so on. These migrating computations do not fall into the class of applications that the substrate supports. However, just as it is desirable to hide as much as possible of the of the substrate’s dynamic handler instantiation from the programmer, it is desirable to hide the details of migrating computations. For this reason, the structure and design of the the substrate can be applied to mobile agents as well.

Extensible Operating Systems: Some of the interpreters for the languages described above provide controlled access to local system services. For more flexible access to these services, several research efforts are focusing on safe, application-specific extensions to operating systems. Inferno [9] focuses on providing a network operating system for code fragments through a uniform global name space. SPIN [2] and Exokernel [8] allow applications to design their own OS abstractions, or extend OSs for increased performance or functionality. the substrate is orthogonal to this work. However, these efforts will result in techniques for safely extending a software subsystem. These techniques can adapted to ensure safe eager handler instantiation in the the substrate actuator.

Activating the Network-Layer: The Active Networks [22] architecture introduces capsules, application data packets containing application-specific
code which is interpreted at intervening routers. Two ongoing efforts focus on realizing such active network-layer architectures. Nscript focuses on a network model and a data-flow language for actively programming routers [24]. The SwitchWare [18] project intends to use standard-ML for programming input and output ports of switches. We view the substrate as being complementary to these efforts; the protocols that the substrate contains for eager handler distribution can be used for activating network-layer fragments as well.

**Modular Composability:** The notion of constructing applications and services by composing reusable code is long-standing practice. Various toolkits, libraries, rapid GUI development platforms, and “visual” languages are testaments to this. The recent Java Beans proposal [12] attempts to standardize the packaging of reusable code modules. The substrate extends these ideas to their logical conclusion: re-use of remotely implemented modules, dynamically retrieved upon reference.

**Protocols for Distributed Services:** Over the past few years, several application-specific protocols have been designed and deployed for multi-party communications. Examples include Scalable Reliable Multicast [10], Real-Time Transport Protocol [17] and Distributed Interactive Simulation [14]. We believe that future distributed services will also develop application-specific protocols for multi-party communications; i.e., network-layer support that meets the varying needs of these applications is unlikely. Our work on the substrate will broaden our understanding of the design of such protocols for large internetworks.

**References**


