A Distributed Object Framework for Pervasive Computing Applications

Hubert Pham
MIT CSAIL, 32 Vassar Street, Cambridge MA, 02139 USA

1. Motivation

Robust pervasive computing applications today span heterogeneous systems and often need to be dynamic and adaptive. However, traditional, asynchronous distributed systems are generally too complex and require developers to be deeply aware of the intricacies of the underlying system and platform. Furthermore, building adaptive applications also proves difficult because traditional systems tend to impose a static API between distributed components. The interfaces are determined at compile-time and provide no mechanism for changing the relationships between components during runtime, such as to adapt to hardware upgrade or failure.

This paper outlines several desirable traits of an adaptive software component architecture and presents a new set of abstractions that streamline development of distributed, pervasive systems. The current implementation forms the basis of the component system in O2S (Saif et al., 2003).

1.1 Requirements

The first requirement is a clean and simple component interface, such that the application logic may modify or construct new implementations when adapting to changing environmental requirements. By separating the programming interface and the implementation technology, the component implementation can still be highly parallel and asynchronous, while a simple interface enables developers to construct, monitor, and debug these implementations.

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

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

2. Architecture

While there are many ways to use existing distributed object packages to fulfill the architectural requirements, this paper explores one abstraction that promotes a separation between policy and mechanism. 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 application 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.

In essence, 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. The application logic also monitors the operation of the resulting circuit via a stream of high-level messages generated by the components. These message streams are used to report component failures, user inputs, or various resource-specific notifications. The health of devices hosting these components is also transparently monitored; component state updates and debugging output are collected, filtered, and serialized for presentation to the application logic. It thus 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.

2.1 System Architecture

The system architecture comprises four components:

**Synchronous Control** A standard network object model (e.g., Remote Procedure Call) provides a synchronous control layer, which forms the basis of the simple environment
for instantiating and connecting distributed modules. 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 data streams bypass the RPC system and hence do not incur the associated overhead. The operation of streams are somewhat autonomous, in that once the network is established, data simply flows between modules. Data streams are designed for applications that depend on routing real-time or rich media data between distributed components. Streams also encourage component re-use by providing the mechanism for connecting together generic components to form new applications.

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

**Discovery & Health Monitoring** Servers hosting network objects can often fail from network, power or hardware failure. The system is designed to detect such failures and inform the appropriate dependencies of the failed network object. The architecture also provides resource discovery, enabling applications to potentially recover from failures by discovering and substituting the failed object for an alternative resource during run-time.

### 2.2 The Resources Abstraction

The Resources abstraction is a versatile RPC framework that facilitates the system design. With many traditional RPC packages, developers must generate client and server stubs for code modules, with some predetermined notion of where (which physical hosts) these modules will run. The amount of manual developer effort renders traditional RPC system unwieldy for dynamic, pervasive environment. The Resources abstraction addresses these issues by providing an extensible and portable network object model that features dynamic stub generation, object interning and reference tracking. One resulting feature is that developers need not know *a priori* where code modules are executed, nor whether procedures are implemented locally or remotely: the invocation API is standardized, and the optimal invocation mechanism is always executed automatically.

### 3. Implementation Summary

As this architecture is platform and language independent, interoperable implementations of the architecture have been developed for a variety of languages (Java, Python, C) and platforms (Windows, Linux, Mac OS X).

The Synchronous Control layer is an instantiation of the Resources abstraction, which makes use of XML-RPC as the underlying transport mechanism – although the architecture is compatible with any modern RPC implementation. Data Streams are implemented as uni-directional TCP connections but represented as a Resource object, allowing developers to control and wire together these streams. Finally, special entities, called Resourceys, monitor health and provide notification and lookup services. Resourceys monitor liveness of objects via periodic UDP tokens and notify other objects whenever their dependencies fail.

### 4. Evaluation

This framework has been used to develop several O2S applications; the abstraction of separating mechanism and policy has simplified distributed application development.

Performance-wise, benchmarks suggest a five-fold cost in using the Resources abstraction for network object calls, when compared to Java RMI (Sun Microsystems, 1994). However, when streaming a 1MB file between hosts, the Data Streams implementation introduce some overhead compared to standard C sockets (two fold) but outperform RMI by a factor of five. The performance analysis is consistent with the architectural goals of the system; while RPC is flexible and convenient for constructing, connecting, and controlling distributed modules, we optimize performance for the high-bandwidth data streams.

### 5. Related & Future Work

Many standard RPC systems, such as Java RMI, CORBA (Object Management Group, 2001), and JINI (Waldo, 1999) subscribe to the conventional distributed application model, where applications are composed of statically-partitioned client-server modules. With traditional RPC derivatives, reusing components to construct different applications is often unwieldy, while adapting distributed applications by replacing constituent components is difficult.

Future work for this project include a security and authentication framework, as well as component hot-swapping for streamlined service upgrade and recovery.

### References


