A Framework for Prototyping J2EE Replication Algorithms*

Özalp Babaoğlu[1], Alberto Bartoli[2], Vance Maverick[1], Simon Patarin[1], Jakša Vučković[1], and Huaigu Wu[3]

[1] Università di Bologna, Bologna, Italy
[2] Università degli Studi di Trieste, Trieste, Italy
[3] McGill University, Montreal, Canada

Abstract. In application server systems, such as J2EE, replication is an essential strategy for reliability and efficiency. Many J2EE implementations, both commercial and open-source, provide some replication support. However, the range of possible strategies is wide, and the choice of the best one, depending on the expected application profile, remains an open research question.
To support research in this area, we introduce a framework for prototyping J2EE replication algorithms. In effect, it divides replication code into two layers: the framework itself, which is common to all replication algorithms, and a specific replication algorithm, which is "plugged in" to the framework. The division is defined by an API.
The framework simplifies development in two ways. First, it keeps much of the complexity of modifying a J2EE implementation within the framework layer, which is implemented only once. Second, through the API, the replication algorithm sees a highly abstracted view of the components in the server. This frees the designer to concentrate on the important issues that are specific to a replication algorithm, such as communication.
We have implemented the framework by extending the open-source J2EE server. Compared to an unmodified server, the framework adds a performance cost of about 22%. Thus, it is quite practical for the initial development and evaluation of replication algorithms. Several algorithms have already been implemented within the framework.

1 Introduction

In application server systems, replication is an essential strategy for reliability and efficiency. For example, there is a substantial literature on the replication of CORBA component servers. Sun's J2EE [1] is a more recent application server architecture, now very widespread. Many J2EE implementations, both commercial and open-source, provide some replication support. However, the range of possible replication strategies is wide, and the choice of the best one, depending on the expected application profile, remains an open research question.

* This work was partially supported by the European Commission under the IST program of the 5th Framework through Project ADAPT (IST-2001-37126).

To design and evaluate a replication algorithm for J2EE (or any practical component architecture) requires a substantial investment in development. To ease this process, we introduce the ADAPT framework for prototyping J2EE replication algorithms. It factors the task of developing a replication algorithm into two layers: the framework itself, which handles all the detailed interactions with the underlying server code, and the specific replication algorithm, which is plugged into the framework. The framework is implemented once and for all, allowing the developers to concentrate on the relevant details of the specific replication algorithm.

1.1 J2EE and Replication

A J2EE application is deployed as a set of *components.* Components are objects whose lifecycle and invocation is managed by the server, rather than directly by the developer. The developer writes deployment descriptors specifying such properties as transactional behavior, security, and persistence. At runtime, a client invokes a component through a special lookup mechanism; the server intervenes in each such invocation, to enforce the rules given in the component's descriptor.

In J2EE, the components are called EJBs ("enterprise Java beans"). The principal categories of EJB are "entity beans", corresponding to objects in the business data model, and to rows in the underlying database; and "session beans", corresponding to client connections. Session beans are further divided into stateless and stateful types.

In addition to client-server applications, we are also interested in implementing web services. J2EE specifies that this may be done by using a stateless session bean as a web service endpoint ([2], section 5.5). Also, Axis [3], a widely-used SOAP engine which forms the basis of our web service support, provides an alternative deployment model. In the Axis model, a web service endpoint is implemented by a special type of component, simpler than an EJB, but likewise deployed with a declarative descriptor. We support both models; the replication algorithm sees them as different component types (Section 2.1).

Fig. 1 shows a schematic J2EE application, consisting of components of several types. The one J2EE component type we have not handled so far is the message-driven bean. (See [2], section 15.) We believe that our framework can be applied to message-driven beans as well, and we plan to add them in a future version.

The main advantage that the J2EE component model offers to the developer is that it can partially automate important aspects of the application's logic. The developer expresses transactional logic, for example, in simple, centralized declarations, rather than in many lines of scattered imperative code.

The component model also has benefits for the development of replication support. It requires the developer to keep application state within clearly-declared objects; the invocations of these objects are intercepted by the server; and transactions are handled by a central transaction manager with a well-known API. Together, these constraints mean that in a J2EE system, all the application events of particular interest for replication are already "exposed" to the

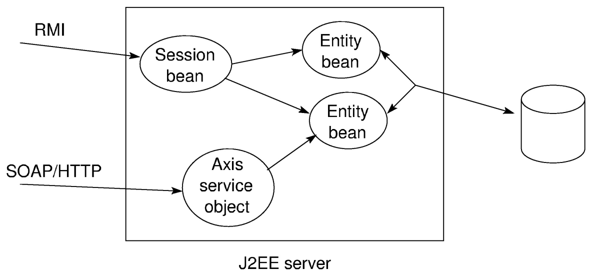

Fig. 1. J2EE components of several types, executing in a single server. Java clients invoke beans through the RMI remote-procedure-call protocol. Web service clients invoke Axis service objects through SOAP messages sent by HTTP. In this example, both these components invoke entity beans, which are mapped to persistent storage.

application server so that, in principle, a replication algorithm can observe and intervene in them.

However, an application server is a complex piece of code, and modifying it is not easy. Further, much of the work of modifying the server is common between different replication algorithms: most algorithms, for example, will need to intercept component invocations from outside the server. Finally, the modifications will need to be redone or re-examined for every new version of the underlying application server.

For all these reasons, we have chosen to develop J2EE replication strategies in two layers (Fig. 2). The lower layer is a common framework; the specific replication algorithm runs on top. The interaction between the layers is defined by an API. Through the API, the replication algorithm sees a simplified view of the components in the system and the invocations between them. It does not see the underlying J2EE implementation.

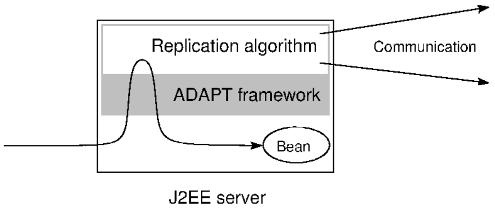

Fig. 2. A replication algorithm is layered on top of the ADAPT framework. When a component is invoked, control passes to the replication algorithm before reaching the component. The algorithm may perform other actions, such as communicating with other replicas, before or after the invocation.

Thus, the specific replication algorithm is a self-contained module, coded more simply than if the server were modified directly. And when a new version of the server is released, the framework is ported forward first; once the framework is ready, specific algorithms can be ported with minimal changes.

1.2 Design Goals

The primary goal of the replication framework is to enable the development of the widest possible range of replication algorithms.

At the same time, it avoids favoring any particular replication technique. Thus, there are some important problems in replication for which the framework does not provide direct support. In particular, it does not specify a model for communication between replicas, or for the notion of a cluster. However, there are widely used abstractions and software packages for handling these problems. Most of the algorithms we have developed, for example, use the group communication model, specifically the JBora group-communication library developed at the University of Trieste [4].

For the same reason, we have disabled the clustering support provided by JBoss [5]. Developers of new replication algorithms are free to adapt JBoss's replication design, and even aspects of its implementation, but they are equally free to develop alternatives.

Finally, another goal has been to remain as close as possible to the original definition of J2EE. For the application developer, in particular, we want to add as little as possible to the requirements already set forth in the J2EE specification.[1] For the developer of replication algorithms, we guarantee that the default behavior of the system—what it does if the algorithm does not prevent it—is the normal behavior of a non-replicated J2EE server.

1.3 Outline

In Section 2, we present the design of the framework, and the elements of the API. Section 3 describes the implementation, based on the JBoss open-source J2EE server, and evaluates its performance. In Section 4 we conclude the paper, discussing related work and some of the replication algorithms that have been developed within the framework.

2 Design of the ADAPT Replication Framework

2.1 Uniform Model of Components

The ADAPT framework API classifies components in three levels:

`ComponentKind` is the broadest classification. There are just a few kinds, fixed by the framework implementation: entity bean, stateful and stateless session beans, web service object. As we add support for other components (such as message-driven beans), we will extend the set of `ComponentKinds`.

[1] As we will see in Section 2.7, we have had to add a few requirements for component state transmission.

`ComponentType` is a kind plus a name, specifying a particular "class" within the kind. The number of types depends on the applications deployed on the server. There is one for each deployed bean and web service.

`ComponentHandle` refers to a specific component instance. It consists of a `ComponentType` plus an instance identifier specific to that type. With an entity bean, the identifier is the primary key; with a stateful session bean, it is the session ID. The number of distinct handles depends on the number of components invoked by the application.

All these classes may be transmitted between replicas. They also support comparison: two `ComponentHandles`, for example, test equal if and only if they refer to the same component instance.

We do not provide access to the underlying Java object implementing the component. Instead, we provide a model of component state (Section 2.7).

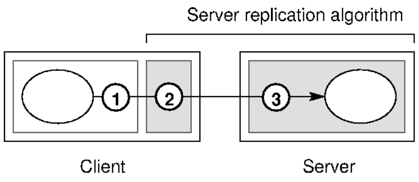

Fig. 3. Key interception points in the course of an invocation between two components. 1. Just before control leaves the caller. 2. The stub: server-specific logic executed on the client side. 3. On the server side, before transferring control to the component.

2.2 Interception of Invocations

In Fig. 3, we show a component invocation, with markers at three key points for replication. At each point, the replication algorithm may intervene, performing any computation or communication before or after continuing. In fact, it does not have to continue execution along this path; it may throw an exception, or return a response computed elsewhere.

1 Just before control leaves the caller. If the caller is replicated, the replication algorithm may synchronize with other replicas before proceeding.
2 The stub, i.e., client-side logic belonging to the server replication algorithm. It may resend requests, fail over to another server host, etc.
3 Just before control is transferred to the target bean. The bean is ready for the invocation, but also for other operations such as reading and writing state.

If the caller is replicated as well as the server, then there are two replication algorithms in this scenario. At point 1, control is still in the domain of the caller's replication algorithm. At point 2, even though execution is still on the calling host, logical control belongs to the callee's algorithm. In particular, if the stub detects a server failure, it can fail over to another server host. The failover mechanism depends entirely on the server's replication algorithm, not the caller's. (We indicate the domains of the two algorithms with shading.)

The replication algorithm expresses the logic at each of these points by implementing an interface. The framework API defines a "local" interface, for logic in the local server, and a "stub" interface, for logic referring to a remote server. When the framework intercepts execution at points 1 and 3, it passes control to the local interface, implemented respectively by the replication algorithms of the client and the server. At point 2, it passes control to the stub interface, which runs on the caller but provides invocation logic for the server's replication algorithm.

In an EJB invocation, the stub interface is actually downloaded from the remote server during EJB lookup. In a web service invocation, though, the client cannot download code from the server (and would not trust it if it did). In a real web service application, server-specific logic on the client side would be spelled out in the service contract, and implemented by the client. For convenience in prototyping, though, we maintain the distinction between the two interfaces even with web services.

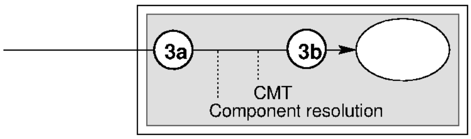

Fig. 4. Within interception point 3, we distinguish early and late stages. At **3a**, the server has just taken control, but it has not resolved the component reference, or set up container-managed transactions, security, or any other EJB invocation properties. At **3b**, the component is ready for invocation.

For EJBs, we distinguish two interception points on the server side, before control reaches the component (Fig. 4). The first point (3a) comes immediately after control reaches the server, before the component reference has been resolved. Breaking here allows the replication algorithm to instantiate the component itself, if necessary. The second point (3b) comes just before control passes to the component—after the reference has been resolved, and all the EJB properties, such as security and transactions, have been set up. At this point, the replication algorithm can get and set the component's state, attach a listener to the transaction, etc.

For web services, there is no useful distinction to be made between these two points, because the invocation model for Axis objects is much simpler than for EJBs. Thus, we provide interception only at 3b.

2.3 Requests and Responses

In the ADAPT invocation API, a `Request` is passed to a component, yielding a `Response`. For example, at interception point 3b, when a component is about to be invoked, the ADAPT framework calls a method in the local replication interface:

```
Response call(ComponentHandle component, Request request);
```

To invoke the component, the replication algorithm calls the corresponding method in `ComponentHandle`:

```
Response call(Request request);
```

Generally, `Request` and `Response` are opaque to the replication algorithm. However, in a `Request`, we allow the replication algorithm to read the name of the method that is being invoked (or, for a web service component, the operation). This permits the algorithm to classify the methods of a component, and treat them differently. For example, if it has access to more information about the application (perhaps an extra descriptor provided by the developer), it might distinguish between read-only and read-write methods.

When the invocation completes normally, the `Response` encapsulates the return value (or, for a web component, the SOAP response). In this case, the replication algorithm cannot examine the content. When the invocation throws an exception, though, this is wrapped in a special `Response` which provides the details of the exception and identifies its source.

Application. The exception was thrown by the component, i.e., by the developer-written code within the application. In this case, the replication algorithm should not examine the exception details, but should simply pass the `Response` back, where it will be handled by the calling component.

System. Thrown by the system or framework, for example when the server crashes. Client-side replication code can catch this and “fail over” to another server before returning to the caller.

Replication algorithm. Thrown by the replication algorithm, presumably from some other point in the chain of invocation. In this case, the replication algorithm is free to examine the exception details and handle them as it chooses.

Headers. Both `Request` and `Response` can be tagged with “headers”. These are arbitrary key-value pairs, which are transmitted along with the content of the message, but they are visible only to the replication algorithm. The key must be a string; the value may be of any class that can be serialized in the invocation.

A common use for headers is to tag each request with a unique ID. This is essential, for example, if the algorithm is to guarantee that each request will be executed exactly once, despite retransmissions and communication failures. Generally, the ID is set by the client-side stub (interception point 2), before the request is sent the first time.

2.4 Transactions

J2EE models transactions with a standard API ([1], chapter 4). This defines a `TransactionManager` or coordinator, which is called by clients, application components, and the server itself, to begin and end transactions and to register participants. And it defines a `Transaction`, which may be associated (one-to-one) with a thread. If so, transactional operations in that thread, such as database access and EJB invocation, are logically contained within the transaction.

To let the replication algorithm follow the association of component invocations with transactions, we provide two framework methods. If the bean uses container-managed transactions, then at interception point 3b (Fig. 4), the transaction will already have been associated with the thread. The algorithm can look it up through the `TransactionManager`. If the component manages transactions itself, through direct calls to the coordinator, the framework notifies the algorithm through a callback.

To track the later commit and rollback of a transaction, the replication algorithm may attach a listener, using interfaces defined by J2EE. One such interface is notified after the transaction has committed or rolled back; another participates in the two-phase commit, so it is notified during the prepare phase as well.

If the replication algorithm wants to intervene more actively in local transaction processing, the framework allows it to "wrap" the entire `TransactionManager`, intercepting every transactional event. It may choose to pass the event on to the underlying `TransactionManager`, or to perform its own distributed logic, or both.

Another approach would have been to open the internals of the `TransactionManager` to the replication algorithm. However, this would have meant choosing a particular implementation. Instead, we chose to interact with the `TransactionManager` only through public APIs; thus, as with a non-replicated server, it remains possible to plug in any valid implementation.

2.5 Component Lifecycle

Components are created by application code, whether directly from the client or indirectly through other components. An EJB is created by a call to one of the `create` methods of its home interface. A web service object is created by the Axis engine automatically, in response to the HTTP request. Either way, the framework notifies the local replication interface after the component has been created, passing it the `ComponentHandle` representing the component, plus any creation arguments. Replication code at another site may repeat the creation, using a `create` method of the `ComponentHandle`.

The client may also look up persistent components (entity beans) by their primary keys. When they are found, the server instantiates them, and then the framework notifies the replication algorithm that they have been instantiated.

When the client deletes a component, the framework notifies the replication algorithm before the deletion takes place. The algorithm may not prevent the deletion, but it can perform any related processing, such as reading state or synchronizing other replicas, before the component disappears.

While an entity bean is in memory, the J2EE server treats it as a cached copy of the corresponding persistent data. If its state is consistent with the database, the server may choose to flush it from memory. The ADAPT framework allows the replication algorithm to block a component from being flushed, forcing it to remain in local memory.

Finally, for stateful session beans, J2EE defines *passivation*, the transfer of in-memory state to storage managed by the server, allowing the component instance to be garbage-collected or reused.[2] From the point of view of the replication algorithm, this is not an important change in the component state: the server is still managing the component, and references to it remain valid. Thus we do not expose passivation to the replication algorithm through the API.

EJB lookup mechanisms. We also allow the replication algorithm to intercept the lookup and instantiation mechanisms of EJB even before the component itself is created.

First, J2EE defines a naming service, JNDI. Each component is registered with the JNDI service of its local server. To find the component, the client connects to the service and looks up the component's name.

We allow the client stub to redirect the JNDI lookup, by intercepting the creation of the JNDI `Context` interface on the client side. The replication algorithm may substitute a `Context` referring to a different server, or even provide a custom implementation of the interface.

Second, the JNDI lookup yields the home interface for the component, which provides methods to create new instances and to find existing ones. The invocations of these methods are intercepted on both the client and server sides, at points 2 and 3 in Fig. 3. On the client side, the replication algorithm may redirect the calls to another server; on the server side, it may perform related actions (such as deployment) before allowing the server to proceed.

2.6 Deployment

A component cannot be instantiated on a server unless it is deployed there, that is, its code, configuration, etc., are available. Before replicating components across a cluster, an algorithm must ensure that they are deployed on all servers.

In J2EE ([1], chapter 8), components are deployed in archive files, with a specified filename extension and internal structure. Typically, this file is "delivered" to a server by being copied to a specified directory. The server checks the directory at startup, and at regular intervals afterward.

Our API provides a simple model for deployment information. Each component archive is represented by an identifying handle and a content object, which can be transmitted together or separately. At startup, the replication algorithm can query the framework for all the units that are currently deployed. During runtime, the framework notifies the replication algorithm whenever a new unit is deployed. The algorithm can transmit the handle to its peers, which can test

[2] The term "passivation" is also used for entity beans, but it refers to the cache flushing mentioned above.

whether the handle is deployed locally. If not, they can deploy it through the framework, by providing the handle and the content object.

2.7 Component State

A correct replicated server must generate the same responses as a non-replicated server, for the same series of requests. To achieve this, one strategy (active replication) is to run the same computation in parallel on all replicas. But many common strategies repeat only parts of the computation. To ensure correctness, they must manipulate the *state* of the servers in the cluster. For example, in a primary-backup scheme, the replication algorithm keeps the state of the backup servers in sync with the primary. If the primary crashes, execution fails over to one of the backups. Since this backup is in the same state as the primary, it can continue processing requests, yielding the same responses.

The framework API provides a view of server state based on components. The state of a server is composed of the state of all the components that are active. The framework provides methods to test whether a component has state, and to get and set the state. In the API, the state of a component is an opaque serializable object, which can be sent between replicas.

To implemente these methods for EJBs, we took advantage of existing contracts between the developer and the container. To get the state of a stateful session bean, for example, the framework uses the passivation mechanism, saving the state to an array in memory instead of persistent storage. For entity beans, the framework reads and writes the attributes defined by container-managed persistence.

For Axis web service objects, however, the existing contract with the developer does not include any state methods. Thus, to support state operations, we add an extra requirement: the service class class must implement the Java `Serializable` interface. The state value is simply the serialized form of the object itself. When the replication algorithm sets the state of an Axis component, the framework replaces the underlying object with a fresh instance.

State transfer. In many algorithms, when a server joins a cluster, one of the other servers transfers a large collection of component state to the new one. Request processing must often be blocked during this interval, on the new member or on the one that is sending state. To support this, we provide `suspend` and `resume` methods, to block and unblock requests on the local server.

2.8 Server Addresses

A J2EE server receives TCP connections at several ports: for RMI invocation, JNDI lookup, and HTTP requests. When a client fails over, it must connect not only to a different IP address, but also to the appropriate ports at the new server.

To make failover simpler and more abstract for the replication algorithm, we supply a `ServerAddress` class that encapsulates the IP address and ports for a server. Within each server, the framework supplies the local `ServerAddress`. The replication algorithm can share addresses between the servers in a cluster, and transmit a set of addresses to the client, using `Response` headers or the serialization of the EJB invocation stub.

For testing, it is possible to run several servers on one host. In this case, their `ServerAddress`es have the same IP address but different ports.

3 Implementation

We have implemented the replication framework by building on the open-source J2EE server JBoss [6]. For web services, we used the SOAP engine Axis [3], which is integrated into JBoss.

In each case, the existing architecture provides hooks for intercepting and restarting invocations. JBoss's EJB implementation is structured around "interceptors", a pattern which is built up into invocation "stacks" described by a configuration file. Axis supports the handler model defined by JAX-RPC [7]. A configuration file defines the sequence of handlers to be executed before a service request is finally delegated to the service object.

In each case, we were able to customize the implementation without modifying the Java source files of the server. For both JBoss and Axis, we modified the configuration files, inserting our own interceptors and handlers into the existing invocation paths. This separation of our code from the server will make it easier to port the framework to future server versions.

3.1 Performance

To evaluate the performance of the framework, we used the ECperf benchmark [8] provided by Sun. The benchmark uses a standard manufacturing / supply chain / inventory problem as an example to exercise the numerous features defined in the J2EE specification. The benchmark consists, for one part, in several bean archives to be deployed on the application server and a database schema to be instantiated and populated, and, for the other part, in a configurable client application that uses those previously deployed beans to stress the server. The main configuration variable is the target *transaction rate* at which the client will try to make the server operate by creating an appropriate number of independent client threads. Once the test is completed, the client software collects statistics from the different threads and extracts an average "business operations per minute" computed over the steady state phase of the run (the ramp-up and ramp-down phases are ignored).

In our experimental setup, the JBoss application server and the PostgreSQL database management system run on a single 2.4 GHz Pentium IV Linux machine with 1 GB of RAM (which is otherwise idle). Clients are executed on a different machine, located on the same local area network.

In this environment, we measured the performance of two configurations. The first was JBoss, without the ADAPT framework (and without the JBoss replication mechanism). The second was the ADAPT modified configuration, with a "dummy" replication algorithm that causes it to behave as a normal non-replicated server. The results of this experiment are shown in Fig. 5.

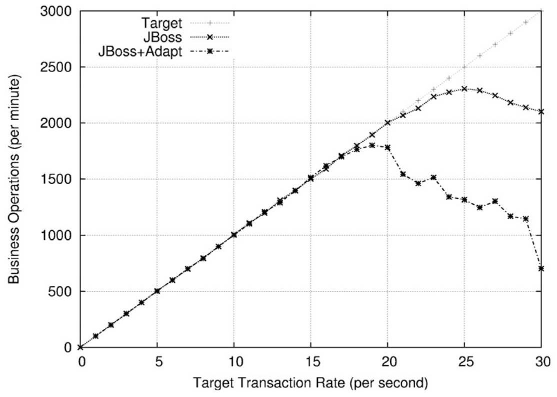

Fig. 5. Experimental results of the ECperf benchmark run against an unmodified version of JBoss and against the ADAPT framework. The *target* line shows expected throughput according to the chosen transaction rate.

The general shape of both curves is the same: the number of operations increases linearly with the target transaction rate until the server becomes overloaded. After this point, some transactions begin being aborted, due to timeout or contention on the database management system. This overload landmark is reached at 21 transactions per second for unmodified JBoss, and at 18 transactions per second for the ADAPT configuration. The throughput maxima are 2105 and 1800 business operations per minute respectively. In other words, the penalty for adding the framework is a 22% drop in throughput. One should also note that beyond the overload landmark, performance degrades more steeply in the ADAPT configuration.

Clearly, adding the framework has a real impact on the server. However, at 1800 ECperf business operations per minute, the performance is still quite practical, enabling developers to run realistic benchmarks against their prototype replication algorithms.

4 Conclusion

We have presented a framework for prototyping replication algorithms for J2EE application servers. We have implemented the framework by modifying the open-source JBoss application server. The modifications add a performance cost of about 22%. We believe this cost is quite acceptable for prototyping, because the development of replication algorithms is much easier within the framework.

If an algorithm performs well in this prototyping environment, developers may consider reimplementing it by direct modification of the server. However, this is unlikely to improve performance by as much as 22%. First, some of the implementation mechanisms of the framework, such as interception of invocations, will be required in a "direct" implementation as well. But more importantly, replication algorithms add other substantial performance costs, such as interserver communication.

Several other recent projects have related goals. Bennani et al. [9] present a replication algorithm implemented with CORBA portable interceptors and Java serialization. We have used similar implementation mechanisms for the framework layer, but built on them to give the replication algorithm a more abstract view of the system. Further, our framework provides some important features which they found lacking in the CORBA interceptor mechanism. In ADAPT, replication algorithms may block the processing of a request, returning a response computed elsewhere. And they are free to create threads and communicate with one another, using all the mechanisms of J2EE.

Marangozova and Hagimont [10] present a model for CORBA component replication. As in our work, they separate replication concerns from the application and from the underlying system. The main difference is that they define replication policies at the level of the component. ADAPT supports a single policy on each server. Since the replication algorithm sees the entire set of components in the server, it can coordinate replication actions such as state transmission across components. (But since it sees the types of the components, it can also define per-component replication policies if desired.)

Several practical replication algorithms have been implemented within the ADAPT framework. Our colleague Milan Prica at the University of Trieste has used the framework to implement a version of the JMiramare algorithm described by Bartoli et al. [11], focusing on web service components. And one of the authors (Wu) has used the framework as the basis for a new replication algorithm [12], focused on session beans, providing strong guarantees of consistency and exactly-once execution.

Version 1.0 of the framework has been released on the SourceForge open-source hosting system, at `http://j2ee-adapt.sourceforge.net`.

References

1. Shannon, B.: Java^{TM} 2 Platform Enterprise Edition Specification, v1.4. Sun Microsystems, Inc. (2003) `http://java.sun.com/j2ee/1.4/docs/`.
2. DeMichiel, L.G.: Enterprise JavaBeans^{TM} Specification, Version 2.1. (2003) `http://java.sun.com/products/ejb/docs.html`.
3. Apache Web Services Project: Axis SOAP library, version 1.1 (2003) `http://ws.apache.org/axis`.
4. Bartoli, A., Prica, M., Antoniutti Di Muro, E.: Reliable communication (2004) ADAPT project deliverable describing the JBora toolkit.
5. Labourey, S., Burke, B.: JBoss Clustering. The JBoss Group. (2002)
6. JBoss Group: JBoss 3.2.3 (2003) `http://www.jboss.org/`.
7. Chinnici, R.: Java^{TM} API for XML-based RPC: JAX-RPC 1.1. (2003) `http://java.sun.com/xml/jaxrpc/index.jsp`.
8. Sun Microsystems, Inc.: ECperf^{TM} specification: 1.1 final release (2003) `http://java.sun.com/j2ee/ecperf/`.

9. Bennani, T., Blain, L., Courtes, L., Fabre, J., Killijian, M., Taïani, F.: Implementing simple replication protocols using CORBA portable interceptors and Java serialization. In: International Conference on Dependable Systems and Networks (DSN-2004), Florence, Italy (2004) `http://www.laas.fr/~ftaiani/ressources/DSN2004-final-cprght.pdf`.
10. Marangozova, V., Hagimont, D.: An infrastructure for CORBA component replication. In: Proceedings of the First International IFIP/ACM Working Conference on Component Deployment (CD 2002), Berlin, Germany (2002) `http://sardes.inrialpes.fr/papers/files/02-Marangozova-CD.pdf`.
11. Bartoli, A., Prica, M., Antoniutti di Muro, E.: A replication framework for program-to-program interaction across unreliable networks and its implementation in a servlet container. Technical report, Dipartimento di Elettrotecnica, Elettronica, Informatica, University of Trieste, Trieste, Italy (2003) `http://adapt.adapt.cs.unibo.it/papers/AntonBartPrica.pdf`. Submitted for publication.
12. Wu, H., Kemme, B., Maverick, V.: Eager replication for stateful J2EE servers (2004) Submitted for publication.