Middleware layers for the *Virtual Nodes* Replication Framework

Jörg Domaschka
E-mail: joerg.domaschka@uni-ulm.de

1 Introduction

The *Virtual Nodes* replication framework is a software tool that helps replicating legacy and new applications easily. Hence, it may serve as an important building block for reliable and available systems. Tailored towards support for distributed services, the framework consists of a client-and a server-side part. Additionally, it exhibits a separation between a replication layer (core) and a middleware layer that mediates between the client application and the replication layer on client-side; and between the replication layer and the service implementation on server-layer.

The entire architecture is sketched in Figure 1. The core of the framework has already been presented elsewhere [1], [2], [3], as have the general requirements towards and implementation of the middleware layers and the message flow. In this document we present the realisation of two different middleware layers on top of the *Virtual Nodes* core layer.

The core functionality of the middleware layer in the *Virtual Nodes* framework differs from client-side to server-side. On the client-side, it has to realise binding and remote access to the service, both in a replication-unaware manner. On the server-side, it has to translate incoming requests to calls to the service instance. Client- and server-side have to agree on a consistent marshalling of request parameters and response values. Furthermore, the *Virtual Nodes* framework requires each middleware layer to define its own method identifiers.

In an extended scenario, the middleware layer has to offer the same context to the service and client implementation as the original middleware would. While it is basically possible to re-implement a middleware layer from scratch that is API-compatible to an existing middleware system, it is more efficient to integrate support for *Virtual Nodes* in an existing middleware system and to tweak the affected parts of the system. A middleware system is *Virtual Nodes*-enabled, if it can be used with fragmented objectssuch as Aspectix [4]. In a nutshell, this requires the possibility to inject code in the client-side stub so that *Virtual Nodes* takes over all communication issues.

In the *Virtual Nodes* design it is the *object adapter* that accesses the middleware layer on server-side via well defined interfaces. We present the object adapter in Section 2. This design reflects the intention that the replication layer shall be independent from any middleware details and vice-versa. Yet, in some cases, it is necessary to feed back information from the replication layer to the middleware layer. The *Virtual Nodes* system introduces *middleware adapters* for that purpose. We discuss them in Section 3.

Starting from the experience of integrating early versions of *Virtual Nodes* in a CORBA environment [5], [6], we successfully implement a middleware layer for Java RMI [7] and the *Distributed XtreemOS Interface (DIXI)* [8]. We briefly sketch these two middleware layers in Section 4 and Section 5.

2 The Object Adapter

The object adapter module is a thin layer of abstraction that isolates the replication protocol module from the middleware layer. This allows modifying the interaction of the replication layer with the middleware layer without having to update all available implementations of replication protocols and vice-versa. A thread that shall invoke the replicated service first passes through the object adapter, from there to the middleware dispatcher, and only afterwards to the service. The object adapter supports fully asynchronous invocation processing. In consequence, a thread that invokes the middleware layer does not necessarily have to produce a reply. Instead, the reply may be triggered by any other thread. This functionality is for instance used in the DIXI middleware layer (c.f. Section 5). The object adapter module controls access to the invocation cache and steers the serialisation of the application state. Yet, it does not perform the actual serialisation.

The replication layer of the *Virtual Nodes* framework has no direct access to the service implementation. In consequence, state serialisation of the service implementation can only happen mediately via the dispatcher in the middleware layer. The same is true for deserialisation. Yet, at deserialisation the replication layer cannot deserialise the dispatcher, because the replication layer is unaware of the concrete middleware implementation it shall be linked to. For that reason, the object adapter stores a middleware-specific serialisation handler. During state serialisation, the object adapter first serialises its own data such as the invocation cache and then delegates the serialisation of the dispatcher module and the service implementation to the serialisation handler. Similarly, during initialisation after state transfer, the object adapter first re-establishes its own state and then uses the handler to create the dispatcher module and service implementation.

The serialisation handler is middleware-specific. Thus, it is the middleware layer that decides on the prerequisites the service implementation has to fulfil with respect to state transfer. Options include using getter and setter methods as required by the CORBA standard, to use Java serialisation for the service implementation, or to allow the definition of application-specific serialisation handlers.

• *J. Domaschka is with the Institute of Distributed Systems, Ulm University, Germany; and with iTransparent GmbH, Nürnberg, Germany.*

originally published November 1, 2011, last revised August 29, 2012. It is currently under further revision.

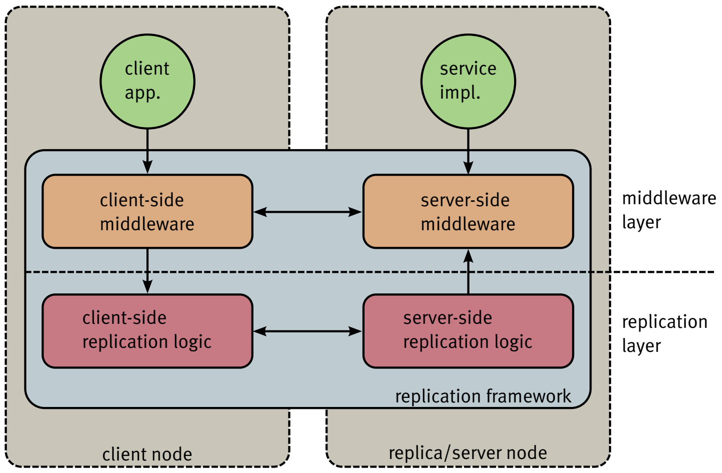

Figure 1: Architecture of the *Virtual Nodes* framework

```
public interface ReplicatedObject {
    public void invoke(MethodId, byte[], ReplyCallback);
    public boolean isReadOnly(MethodId);
}
```

Figure 2: Interface of `ReplicatedObject`

Implementation: The object adapter communicates with the middleware layer using the interface shown in Figure 2. `ReplicatedObject` contains two methods, one for *middleware dispatcher* and one for the *method repository*.

`invoke(...)` calls the dispatching module. The dispatcher executes parameter unmarshalling, method identifier mapping, and invokes the service implementation. `MethodId` denotes which method to call, while `byte[]` contains the serialised parameters. By the method information, the *dispatcher* determines how to interpret the information in the `byte[]` parameter. `ReplyCallback` is a callback handler from the replication protocol. The middleware layer has to use it to trigger the processing of the reply.

`isReadOnly(MethodId)` provides access to the *method repository* module and allows the replication layer to retrieve information about the read-only characteristics of the methods.

3 The Middleware Adapter

In many object-oriented middleware systems object references to remote objects may be passed from one node to another. Knowing the middleware system being used, a node that receives such a reference is able to interact with the remote object referenced by the remote reference. Basically, there exist two ways how a middleware layer may serialise its object references. The *serialisation approach*, e.g. used by Java RMI, serialises the stub object at sender-side. At receiver-side, the stub is re-created by deserialisation. All information that was contained in the stub object at the sender, is also contained at the receiver. The *mapping approach*, applied by systems such as CORBA and ICE[1], uses an external object representation; *IOR* in the case of CORBA. Whenever a stub object is to be serialised, the middleware runtime maps it to an object reference. At the receiver, the middleware runtime interprets the information contained in the reference and creates a stub object from it. Both approaches work fine when the object reference is static and does not change.

In the replicated scenario the replica group changes over time due to replica failures and restarts. Jointly with the replica group, the object reference has to change. In the serialisation approach the object reference changes automatically when the state of the stub object changes. Hence, the object reference is always up-to-date as long as it is contained in the stub object. In the mapping approach, the middleware runtime may have to change the mapping of a stub object to the object reference according to the composition of the replica group. In turn, this is only possible, if the middleware runtime notices changes of the replica group.

In the *Virtual Nodes* framework, only the replication layer is responsible for managing the reference to the replica group. In order to support middleware systems that follow a mapping approach, the replication layer requires a mechanism to inform the middleware system that its internal state shall be updated. Clearly, the implementation of such a mechanism cannot be generic for all middleware systems, but is dependent on how the middleware stores its mappings and assembles its object references. For that reason, we introduce an adapter entity, that is capable of transforming group changes to updates in the middleware

1. http://zeroc.com/

runtime. The adapter is middleware-specific and installed at the initialisation of the replication layer. It keeps object references up-to-date and at the same time replication and middleware layer well separated.

Even though our argumentation so far focusses on the client-side, the adapter is required on the server-side as well, because the server may have to pass an object reference to the client. This is, for instance, the case when a method invocation to the service object returns `this`. Instead of marshalling the service object, the middleware layer has to return an object reference. At server-side, the middleware adapter is managed by the membership module. An extensive discussion of the adapter is subject to earlier work [6].

Implementation

An adapter has to implement the `MiddlewareAdapter` interface. The replication layer will invoke the `newMemberList(MemberList)` method at the `MiddlwareAdapter` every time the replica group changes. The adapter has the task of transforming the information of the `MemberList` to a middleware-internal data representation and inject it into the middleware runtime. Thereby it not necessarily required that the core of the middleware system can interpret the data, it is only required that the data be transferred in an opaque way.

4 Java RMI Layer

Java RMI is the remote method invocation mechanism provided by the Java specification. Thus, it is the first choice for distributed Java programmes. In the following paragraphs we first show how Java RMI works. Then, we present the Java RMI layer for the *Virtual Nodes* framework. Finally, we discuss differences and open issues.

4.1 Java RMI

Java RMI allows a programmer to *export* any object in the system. The only restriction is that this object implements the `RemoteObject` interface. All methods of that object that stem from an interface extending `RemoteObject` are available via remote method invocation. During the export process, the RMI runtime creates a stub object for exported object. The stub contains every information required to contact the exported object. Thus, in order to access the object from a remote host, it is required to copy it to that host. Java allows any kind of propagation to copy the stub. The suggested way, however, is to use the RMI registry. When exporting the object, the programme may also want to start a registry on the local machine and store the stub in that registry using a named key. Applications using the stub can connect to the registry and retrieve a copy of it.

There are two possible ways to create the stub class. The first one is to statically generate it using the *rmic* compiler. The second one is creating it on the fly. Since, Java 5 the latter approach is the preferred one. Dynamic creation makes use of the Java dynamic proxy. This special class can be fed with a number of interfaces and creates an object that implements all of these interfaces and contains all their methods. All calls to any of these methods end up in an *invocation handler* object. For RMI there is a special invocation handler, `RemoteInvocationHandler` that has to be initialised with details on the location of the object. When exporting an object, the exporter analyses all available interfaces and chooses the ones that extend `RemoteObject`. Then, it creates a unique identifier for the object and initialises a `RemoteInvocationHandler` with that data. Finally, it creates a dynamic proxy using the selected interfaces and the create handler.

When a RMI invocation is to take place, this is only possible if all parameters and the return type are serialisable. Accordingly, they are serialised and then transferred to the server-side, together with the object identifier and a method identifier. Java RMI identifies methods according to their name and parameter list. On the server-side, the RMI framework searches the matching object, deserialises the parameters, and uses reflection to invoke the method. The return value takes the opposite direction. It is important to notice that all data is transferred by-value. That is, if the remote object changes the state of the parameters, these modifications are not reflected on the client-side. There are two exceptions to that rule. When caller and callee reside within the same JVM the invocation does not copy any values. Furthermore, if a parameter or return value is itself exported as a remote object, it is not passed by-copy, but by-value.

4.2 Overview

For realising an RMI-compatible middleware layer on top of the *Virtual Nodes* replication layer we provide a customised exporter that creates dynamic proxies with a replication layer-aware invocation handler. All dynamic proxies implement the `Remote` interface, so that it can be added to RMI registries. For marshalling and method identification we use the same approach as the standard RMI implementation. Similar to Java RMI, the dispatcher on server-side uses reflection to invoke the service. Our implementation does not require the use of a middleware adapter, as all relevant information is contained in the invocation handler. The exporter is only used to create the first replica. All other replicas are created using the administration interface.

Besides the standard RMI features, the RMI middleware layer comes with a number of extensions. In addition to remote services, our stubs support so-called local objects, that implement functionality at client-side. This is similar to AJAX web technology and implements functionality comparable to fragmented objects. Furthermore, methods in the service implementation can be declared to be *ignored*. As a consequence, they cannot be accessed remotely. Moreover, methods in the service implementation can be declared to be *hidden*. As a consequence, they cannot be accessed from clients, but from within the stub and the local object.

Remote Methods: In contrast to the RMI system, we do not require that an exported service implements a dedicated interface. This allows to export arbitrary objects without modifying the code. In particular, existing RMI objects can be exported using the *Virtual Nodes* framework.

```
public interface LocalObject<T> extends Serializable {

        public void setProxy(PrivateProxy _prox);
}
```

Figure 3: Local Object

```
public static final<T> Object
        exportObject(Object service, LocalObject<T> local) throws ExportException;
```

Figure 4: RMI Exporter

Local Objects: A local object is logically part of the service, but physically part of the stub. It is useful to implement local functionality that is useless at server-side such as encryption, caching, or access to the local file system. Objects that shall be used as local objects have to implement the interface `LocalObject` shown in Figure 3. We require that any local object be serialisable. Hence, it is possible to send it to other hosts. Unlike remote objects, local objects can never be transparent to their developer, so that it is not necessary to provide a simple marker interface. In addition, a local object might just do some preprocessing and then call the remote part of the service. For such a scenario the local object has to have a reference to the request handler that invokes methods at the server-side. In our approach this reference is realised indirectly via a `PrivateProxy`.

We allow only one local object per stub. This is not a restriction *per se*, as the development of such an object is not transparent. Hence, this approach does not limit the developer, as it is still possible to use one local object as a façade [9] to a set of other local objects. On the other hand, it significantly eases the implementation of the exporting process.

Exporter Class: Exporting an object with the *Virtual Nodes* framework is similar to Java RMI. An object is exported via the `Exporter` class. During the export process two things happen. First, the `Exporter` initialises an instance of the replication framework that wraps the service to be replicated. Secondly, it creates a stub that provides access to the service.

`Exporter` has a single public method `exportObject` (c.f. Figure 4). It takes a `Configuration` an `Object`, and a `LocalObject`. That is, an object that can be exported, and a `LocalObject`. If exporting is not possible, it throws an `ExportException`, otherwise it returns a stub object. All action that is described in the following happens within this method.

The `Exporter` generates a stub interface from the object. By default it uses all interfaces of both remote and local object and creates a dynamic proxy. In addition, it adds `ClientAdmin` to the proxy interface. That is, all administration methods are directly offered to the user of a service. From an invocation to the proxy object, the handler has to be able to figure out which method invocations are remote method invocations, which of them concern administration methods [1], and which are for the local object. In order for the mapping to be unique, we impose the following restrictions on the method interfaces.

First, local and remote objects must not implement the same interface unless it is a marker interface. Methods that conflict with the administration interface are not allowed in either object. That is, the remote object must not override methods from the `ClientAdmin` or `ServerAdmin` interface. The local object must not override methods from the `ClientAdmin` interface. Finally, remote and local object must not contain methods with the same signature. The `Exporter` extracts all interfaces, checks all these conditions and creates an instance of `ReplicationHandler`. From the interfaces and the handler, it creates a dynamic proxy that is returned to the caller.

State Transfer: In order to support the state transfer we use the following approach. If the exported object is serialisable, and the user does not provide a serialisation handler, as the middleware layer provides its default serialisation handler. The default handler uses Java serialisation to serialise the object prior to state transfer. If the object is not serialisable the `Exporter` enforces that a serialisation handler be installed. If standard serialisation does not work, the deployer of the service can specify a serialisation handler. The handler is saved in the stub object and copied from one node to another. Hence, the handler has to be serialisable itself.

Dispatcher: The dispatcher implementation is straight forward. Parameters are de-serialised and the return value is deserialised just as in the RMI implementation. Here, two issues require special care. If a parameter is a stub to the service to be invoked, it has to be replaced with the service reference. Similarly, if the service reference is to be returned to the client, it has to be replaced by a stub object.

In contrast to the RMI implementation the dispatcher does not have to check whether a stub parameter has its object exported in the same JVM or an return value is another exported object. These cases cannot happen in the current *Virtual Nodes* prototype, as only one exported object per JVM is possible.

We realise read-only methods by annotations. Every method in the exported object that is annotated with `@ReadOnly` in the service implementation yields read-only behaviour at replica-side. The dispatcher uses reflection to inspect the object's methods and to check for the annotation. This is done after during export or after state transfer. The information is then cached in the dispatcher implementation.

Ignoring Methods: The restriction imposed on the methods contained in local object and remote object allow to provide useful services. However, they render several behaviours more cumbersome than necessary. In the following we discuss such situations and provide a solution.

We require that the local object be serialisable. At the same time we provide serialisation support for the remote object, if it is serialisable as well. As `Serializable` is a marker interface this does not impose any restrictions. However, this does not hold for `Externalizable`. This interface adds two methods to `Serializable`, so that it is not a pure marker interface any more. Hence, problems arise when both local and remote object implement it. Obviously, the methods provided by `Externalizable` are not intended to be used remotely. Their sole purpose is to allow serialisation which is a purely local event.

There are two approaches to solve this issue. The first one is to filter `Externalizable` by some hard coded filter rules. This approach is straight forward and does not burden the programmer. On the other hand it is very inflexible, as it does not allow for an easy extension when other interfaces with similar functionality appear. An example of such an interface could be some local monitoring entity.

Consequently, we decided to go for the second approach which uses Java annotations. We do allow methods in both locations to be marked by `@Ignore` which means that the interface they are defined in will not appear in the proxy interface. Marking does not happen within interfaces or for entire interfaces, but happens per method of the implementation of either local or remote object. We chose this approach, because of several reasons: interfaces of the Java library cannot not be used if methods have to be annotated in the interface. Furthermore, interfaces cannot be re-used in another context once they are annotated, if interface definitions are subject to annotations. And finally, multiple interfaces can define the same or equal methods with different annotations. This would lead to conflicts, even if the implementation is most certainly unambiguous. An important consequence of the way marking happens and the way proxies are constructed, is that all other methods defined in the same interface as a `@Ignore` method do have to be `@Ignore`d as well. If this is not the case an exception is thrown during exporting.

Note that this does not effect the superinterfaces of the interface. In particular, it is allowed that two non-marker interfaces `B` and `C` extend the same non-marker interface `A` with the methods of `B` being `@Ignored` and those of `C` are not. This is the case, because it is still possible to add `C` to the proxy interface. In contrast, using `@Ignored` for the methods of `A` would require to `@Ignore` all methods of both `B` and `C`.

Hiding Methods: The existence of local objects brings up the issue that methods at server-side shall be invisible for the clients, but visible for the local objects. This is similar to the administration interfaces where `getInitialisationCredentials()` is only visible for the replication layer at client-side, but not for the client application. Assume, for instance, a local object that implements a cache for a sub-set of methods. If the data is available at client-side it shall be used, if not, it shall be loaded from the server. In consequence, the client application must not have the possibility to bypass the local caching mechanism. In the system presented so far, the only way to ensure that this is not possible, is to use the `@Ignore` annotation and exclude the method from the proxy seen by the client. On the other hand, the local object has to have a way to call the server.

This might happen using the interface offered by the replication layer. Yet again, this is a bad idea, as it would require the developer of the local object to know about (de)serialisation and construction of message identifier and would require to change all local objects when the interface to the replication layer changes. Therefore, we create a *private proxy* for the local object, whereas the client application uses a *public proxy*. Both proxies may differ in the methods they contain. This functionality is realised by a new annotation for the service implementation. Annotating a method as `@Hidden` removes it from the public proxy, but leaves it in the private proxy used by the local object. Apart from their modified visibility properties, the same rules hold for methods annotated with `@Hidden` and those annotated with `@Ignore`.

4.3 Discussion

Figure 5 sketches the architecture of the RMI middleware layer and presents how it is linked with the *Virtual Nodes* replication layer. The arrows denote invocation sequences. The client application is bound to a dynamic proxy object. This proxy offers all public interfaces provided by the local object (light green), the remote object (dark green), and the administration interface (red). All invocations on the proxy result in a call to the *Virtual Nodes*-aware *invocation handler*. This entity dispatches calls to the local object, the replication layer, or the administration layer. The local object has a private proxy. It offers access to the public and hidden (blue box) methods of the of the remote object and access to the administration methods. It does not contain the interfaces of the local object, as those can be referred to using `this`. As both proxies automatically implement the `Remote` interface, they can seamlessly be put into RMI registries.

At server-side calls to administration methods are filtered by the *client thread* [1]. Otherwise, they reach the `object adapter` module and from there the dispatcher in the middleware layer. The dispatcher uses reflection to dispatch to the method call.

All interaction with the RMI middleware layer happens synchronously. That is, a client application blocks for the invocation either to return or to throw an exception. Moreover, the middleware dispatcher blocks until the return value of the invocation is available. In case the method throws an exception, the exception is treated as the reply and sent to the client where it is re-thrown. Further details on the RMI layer including serialisation support, exporting, and marshalling are subject to the technical report [10].

Clearly, the requirement to add annotations violates the requirement of replication unawareness. In order to avoid having to modify the implementation service object, it would be beneficial to support a configuration-based notation of hidden, ignored, and read-only methods. The support for persistent configuration in the *Virtual Nodes* framework allows an easy and transparent implementation for that mechanism.

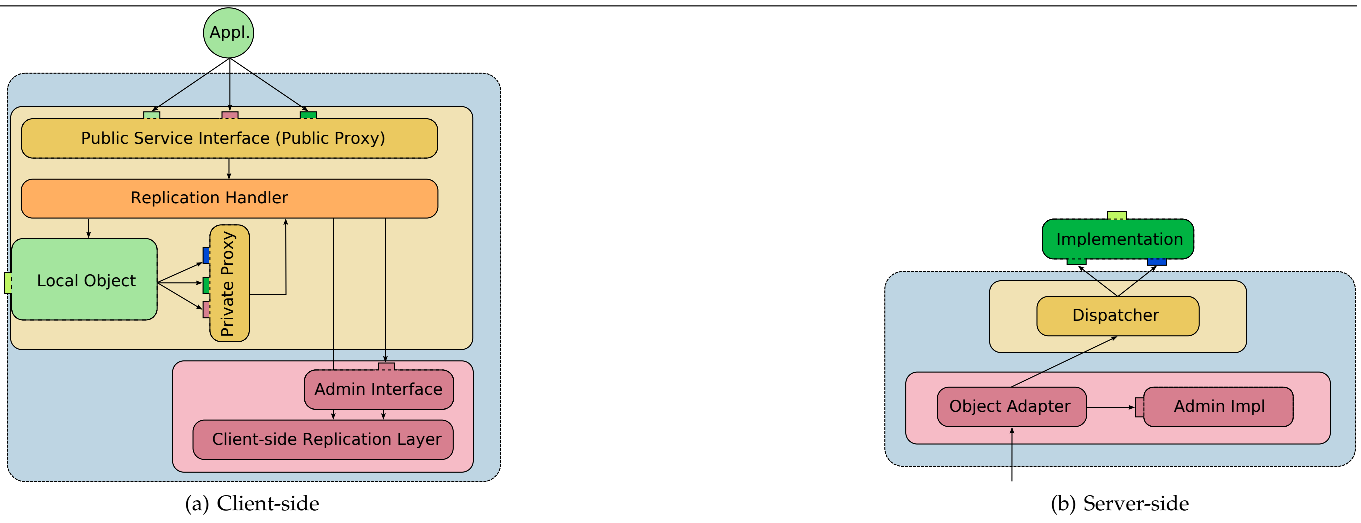

Figure 5: RMI Layer Architecture

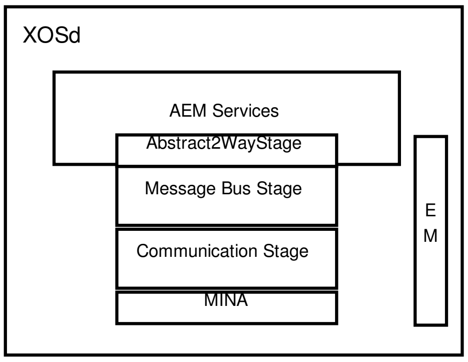

Figure 6: A view of the SEDA stack inside the XOS Daemon

5 DIXI Layer

A central design element of the XtreemOS grid operating system [11] are the so-called core nodes. One of the basic tasks of a core node is to manage the central infrastructure of the grid including the execution and monitoring of jobs and the reservation of resources. In XtreemOS, all those tasks are executed by the *Application Execution Management (AEM)* daemon. Due to their central role in the system, the services that run on core nodes must not fail. Thus, the use of replication is beneficial for them. In particular this is true for the *Job Manager* and the *Reservation Manager*.

The implementation of AEM is based on the *DIXI* library [8] .

In the following paragraphs we first present an overview on AEM, DIXI services, and the DIXI communication infrastructure. Then, we discuss the challenges for integrating such a system with the *Virtual Nodes* framework. Finally, we conclude with a report on the current status and future tasks to be carried out.

5.1 AEM and DIXIi Overview

DIXI [8] is based on the Staged Event Driven Architecture (SEDA) pattern [12]. The core of DIXI is a stage machine that ties together the stages running inside a single DIXI instance. Furthermore, DIXI defines an endpoint per DIXI instance that can be used to send messages to services running in that instance. Messages in DIXI are transferred between stages as event objects. AEM is a DIXI instance with a pre-defined set of stages and services and a particular wiring. Figure 6 shows the standard set-up of a SEDA stack inside an AEM instance. The key property of AEM is that stages are organised in a star topology with the Message Bus Stage as the centre. The *Message Bus Stage (MBS)* functions as a central message dispatcher and router within a single DIXI instance. Messages and events that have to be transferred to other instances are routed to the Communication Stage. In turn, the Communication Stage also receives messages and relays them to the MBS.

Even though DIXI allows stages to implement arbitrary communication patterns every stage implementation used in the AEM uses a request-response pattern. That is, for each ingoing request event, a reply event is generated. Yet, requests are not the only events that reach a service stage. Replies to nested invocations are processed as if they were requests. Yet, they do not generate a reply event.

Even though the communication in typical SEDA systems is asynchronous and hence very loose, all services in the AEM stack provide certain properties. In particular, the reply event to a request event is not generated before all replies to nested invocations have been processed. Furthermore, clients block until they receive a reply. Thus, a client never has two concurrent requests. Messages (events) are characterised by an identifier that is created as a random number. Corresponding request and reply events carry the same identifier.

5.2 A *Virtual Nodes* Middleware Layer for AEM

Integrating DIXI/AEM into a *Virtual Nodes* environment is at the same time integrating *Virtual Nodes* into a DIXI/AEM environment. This is due to the fact that DIXI does not provide a clear separation between communication and middleware functionality. Furthermore, being a container that hosts multiple services, the replication of one of these services shall not have any effect on other non-replicated services. In the following we sketch the architecture of our integrated approach. We only present implementation details where required. The realisation of a transparent integration is aggravated by the fact that messaging and addressing in DIXI does not happen transparently.

Addressing and Messaging: DIXI does not exactly support the concept of a remote method invocation. Instead, messages are send and received. The semantics of whether the message is a request, a reply, or something different is implied from the service implementation. For our integration we exploit the fact that all services run in AEM use the request-response interaction pattern. Further, all requests use `ServiceMessage` as event type, while replies use `CallbackMessage`. This coincidence allows us to logically map DIXI message types to *Virtual Nodes* message types.

DIXI does not support the concept of multicast addresses. Yet, it defines an `Address` interface that is implemented by the `SingleAddress` class. However, all elements of the system only make use of the interface type, so that we can simply introduce a new implementing class `ReplicaAddress` that contains the `SingleAddress` of multiple nodes.

Replication Stage: As the MBS is the central entity in the system and concerned with message routing, we extend its implementation by sub-classing and make the sub-class replication aware. We call the sub-class *Replication Stage*. Whenever the Replication Stage receives a message whose target address is a `ReplicaAddress` it hands the request to the *client interface stage*. Furthermore, the Replication Stage provides mechanisms for state transfer and initialisation of the system.

Client Interface Stage: The client interface stage is a system-wide stage that handles all requests to replicated entities. Therefore, it keeps a `ClientBase` per service. It dispatches incoming request to the respective `ClientBase` and dispatches the received reply to the MBS.

Service Access: As DIXI nodes communicate exclusively using the Communication Stage, we implement a custom external communication module that is capable of receiving DIXI messages and transforming them into *Virtual Nodes* messages. In order for the communication module to be able to receive messages it has to be registered as an own stage in the DIXI stack. Similarly, the communication module takes replies from the *Virtual Nodes* runtime and sends them as DIXI replies to the caller of the method.

Dispatcher: Similar to the RMI dispatcher, the DIXI dispatcher has the task to transform an incoming *Virtual Nodes*-message to DIXI message the service can use. Yet, due to the staged architecture the DIXI dispatcher has additional tasks.

First, the service is a stage on its own. In consequence, it comes with its own threads and interaction can only happen via its income queue. Thus, in order to realise consistency of service stage, the stage either may have only one thread processing updates or the deterministic scheduler has to schedule the stage threads as well. In a first step we let the dispatcher configure the service stage such that it only uses a single thread and configure the system to perform sequential scheduling. In a second step, we extend the scheduler to support priority transfer between threads.

Second, the dispatcher is responsible for intercepting all events generated by its service. For that reason, it functions as an outgoing queue for the service. Inspecting the kind of event, it can decide whether to generate a *Virtual Nodes* reply or a nested invocation. For nested invocations local information has to be stored before the message is re-directed to the Replication Stage.

Third, replies for nested invocations have to be treated special. As all replies are executed as requests, the reply has to be injected in the system when it is received. This task is executed by the dispatcher. Yet, as replies may be available before the request was sent, the thread executing the reply, has to be blocked until the request was sent.

The flow of messages and data for a single request is shown in Figure 7. As one can see, messages arrive at the Replication Stage using the regular DIXI stack. Here, they are not directly handed over to the service implementation, but have to pass the replication infrastructure. In the DIXI receiver the request is wrapped into a *Virtual Nodes* message and relayed to the replication protocol. From there, it eventually reaches the middleware layer (dixi dispatcher) where the DIXI message is unwrapped and then inserted in the in-queue of the service implementation.

Nested Invocations to External Services: Nested invocations addressed to replicated entities are handled by the *Virtual Nodes* infrastructure and the implementation of the dispatcher. However, in DIXI/AEM environments it can happen that a nested invocation targets a singular service. This is due a special capability of the system architecture. Jobs, for instance, are run on resource nodes where replication is not applicable. If the node crashes, the job is lost. For that reason, our prototype implementation requires that such invocations do not fail.

System Set-up: Figure 8 sketches the communication stack that is used in replicated scenarios. The Replication Stage handles messages that target replicated as well as singleton services. Messages that are not directed towards replicated services just pass

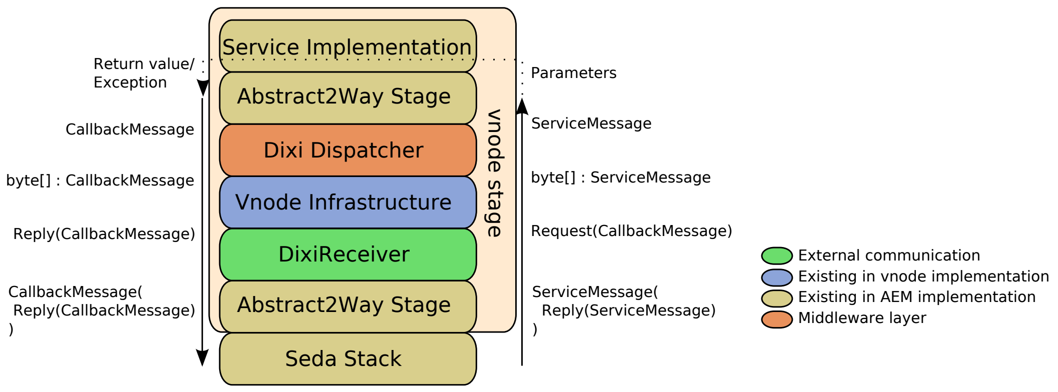

Figure 7: Message and data flow in an integrated version on server-side

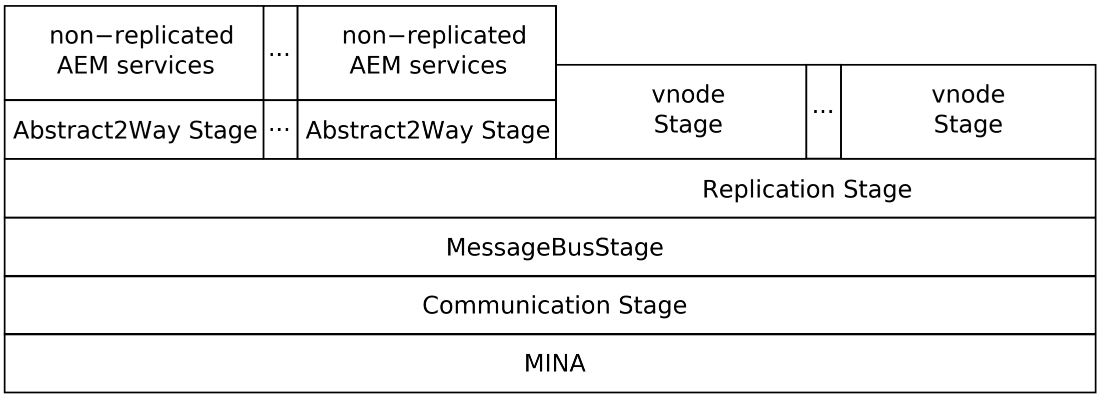

Figure 8: Modified communication stack for replication support

through this stage without any modifications. For messages directed to replicated services there are two options. If the message is issued by a client, and thus not yet replication aware, it is passed on to a replica stub stage where the message is modified to fulfil replication requirements. In any case, it is relayed to Dixi receiver stage of the respective service. Of course the approach of handling client requests is only resilient to node failures when the request origin from local clients. This is ensured by the way an AEM-extension that treats addresses of replica groups. Requests from remote hosts are directly forwarded to their respective stage.

5.3 Discussion

This section has shown how *Virtual Nodes* can be integrated into a staged environment. Due to the fact that DIXI/AEM is research software and lacks essential features of a message-passing middleware such as error detection and re-sending of messages, the current prototype that results from the integration efforts is not usable in erroneous environments.

Nevertheless, due to the integration we gained some positive insights on the *Virtual Nodes* framework. The strong dependency of DIXI/AEM on nested invocations has proven that the capabilities provided by *Virtual Nodes* are robust and generic enough. Supporting the staged design added functionality to the deterministic schedulers. Yet, it has also shown that *Virtual Nodes* are relatively heavy-weight when multiple replicated instances shall be supported within a single virtual machine. Here, it would have been beneficial to have only one instance of the framework for all replicated services. We consider that issue as future work.

6 Summary

In this document, we have presented the middleware layer of the *Virtual Nodes* framework. We have discussed how it is tied to the replication layer via the object adapter module and the way the middleware layer can influence state transfer. We further presented middleware adapters as an approach for the middleware layer to retrieve information about group changes. For some middleware systems, this is an essential feature to keep object references up-to-date.

Afterwards, we have sketched the implementation of two different middleware layers. One to wrap the synchronous, RMI-based Java remote method invocation infrastructure. The other one to provide replication for the core services running on an XtreemOS core node. The DIXI middleware and communication framework is stage-based and widely asynchronous.

Realising replication for two such different middleware systems with the same replication logic is beyond everything found in related work. The fact that it has been possible to seamlessly integrate replication support into both systems shows that the initial design goals of the *Virtual Nodes* replication framework have been satisfied. Furthermore, it is a strong indication that operation-driven replication in invocation-oriented systems is orthogonal to middleware concerns and client applications.

REFERENCES

[1] J. Domaschka, "A comprehensive and flexible approach to transparent replication of java services and applications," Ph.D. dissertation, Fakulät für Ingenieurwissenschaften und Informatik, Universität Ulm, Germany, 2012, submitted.
[2] J. Domaschka, C. Spann, and F. J. Hauck, "Virtual nodes: a re-configurable replication framework for highly-available grid services," in *Middleware (Companion)*, F. Douglis, Ed. ACM, 2008, pp. 107–109.
[3] J. Domaschka, "Extended version of a Virtual Nodes system," Institute of Distributed Systems, Ulm University, James-Franck-Ring O-27, 89069 Ulm, Germany, XtreemOS Deliverable D3.2.14, 2009.
[4] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier, "AspectIX: A middleware for aspect-oriented programming," in *Workshop ion on Object-Oriented Technology*, ser. ECOOP '98. London, UK: Springer-Verlag, 1998, pp. 426–427. [Online]. Available: http://dl.acm.org/citation.cfm?id=646778.706221
[5] H. Reiser, R. Kapitza, J. Domaschka, and F. J. Hauck, "Fault-tolerant replication based on fragmented objects," in *Distributed Applications and Interoperable Systems*, ser. Lecture Notes in Computer Science, F. Eliassen and A. Montresor, Eds. Springer Berlin / Heidelberg, 2006, vol. 4025, pp. 256–271.
[6] J. Domaschka, H. P. Reiser, and F. J. Hauck, "Towards generic and middleware-independent support for replicated, distributed objects," in *Proceedings of the 1st workshop on Middleware-application interaction: in conjunction with Euro-Sys 2007*, ser. MAI '07. New York, NY, USA: ACM, 2007, pp. 43–48. [Online]. Available: http://doi.acm.org/10.1145/1238828.1238839
[7] S. M. Inc, "Java remote method invocation specification," 2004.
[8] M. Arta, "Distributed XtreemOS infrastructure (DIXI)," XLAB d.o.o., Pot za Brdom 100, SI-1000 Ljubljana, Slovenia, XtreemOS Deliverable D3.2.17, March 2010.
[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
[10] J. Domaschka, "Middleware layers for the virtual nodes replication framework," Institute of Distributed Systems, Ulm University, Germany, Tech. Rep. VS-R16-2011, December 2011.
[11] M. Coppola, Y. Jégou, B. Matthews, C. Morin, L. P. Prieto, O. D. Sánchez, E. Y. Yang, and H. Yu, "Virtual organization support within a grid-wide operating system," *IEEE Internet Computing*, vol. 12, pp. 20–28, March 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1399091.1399259
[12] M. Welsh, D. Culler, and E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 230–243, October 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502057