Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Distributed XtreemOS Infrastructure (DIXI)

D3.2.17

Due date of deliverable: March 30^{th}, 2010
Actual submission date: March 30^{th}, 2010

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP3.2
Task number: T3.2.6

Responsible institution: XLAB
Editor & and editor's address: Matej Artač
XLAB d.o.o.
Pot za Brdom 100
SI-1000 Ljubljana
Slovenia

Version 1 / Last edited by Matej Artač / March 26^{th}, 2010

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	15/02/10	Matej Artač	XLAB	Document outline
0.1	22/02/10	Matej Artač	XLAB	Initial import from the technical document
0.2	23/02/10	Matej Artač	XLAB	Architecture
0.5	24/02/10	Matej Artač	XLAB	Anonymous call-backs, usage of XServiceProcessor, Configuration
0.8	28/02/10	Matej Artač	XLAB	Executive summary and Conclusion
0.85	05/03/10	Ian Johnson	STFC	Comments and suggestions
0.85	23/03/10	Matej Artač	XLAB	Integrated internal reviewers' comments
1.0	26/03/10	Matej Artač	XLAB	Final version

Reviewers:
Ian Johnson (STFC) and Ramon Nou (BSC)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.2.6	Distributed XtreemOS Infrastructure (DIXI)	XLAB*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Contents

List of Figures

List of Tables

Executive Summary

The DIstributed XtreemOS Infrastructure (DIXI) has been designed as a framework and a message bus to offer rapid prototyping of XtreemOS system services. DIXI provides a staging environment for the XtreemOS services, simplifying the development of a distributed system by providing a convenient communications layer. This layer can be used to allow services to communicate with each other, whether they are on the same node or different nodes. The DIXI developers designed the system to address both short-term goals, providing a simple way to experiment with new XtreemOS components, and long-term goals, including the deployment of the system on a high number of nodes. This deliverable focuses on describing the framework and providing the details on how to develop services to be run within the framework.

The development of the DIXI was further motivated by the requirement that the service middleware would be light-weight, provide the service bindings in Java, and extend the client bindings with C. The framework should abstract away the particulars of the locations of peer services and the means of communication required to invoke them. The resulting framework thus functions as an integral part of XtreemOS, hosting its vital services, and also providing a bridge between the services hosted by DIXI and other XtreemOS services.

The XtreemOS runtime includes the DIXI daemon which hosts the services, and the libraries that can be used by client programs. In essence, the DIXI architecture consists of the messaging bus, an event machine and the stages. The stages represent the instances of the service implementation. When a service invokes another service's method, the framework queues a service message with the message bus. The event machine retrieves the message and puts it on the target service's private message queue, effectively executing the service method. In this respect a bare architecture addresses the stage communication within the DIXI daemon's process. Special built-in stages extend the connectivity towards other instances of DIXI using TCP/IP or SSL. The design of the networking communication permits passing messages for multiple stages using a single channel, which not only simplifies the point-to-point communication within a subnetwork, but also enables routing them through firewalls and between NAT-based subnetworks.

The XATI and XATICA libraries provide the service call stubs to the programmers developing in Java and C, respectively. The architecture of these libraries also includes the message bus and the event machine, but it runs no stages except for the one that provides the network communication.

To develop a DIXI service, the programmer implements a Java class, then decorates it with annotations provided by the DIXI developer library. The service's methods that are to be exposed, can receive annotations for having them also exposed towards the client programs (XATI and XATICA). Special annotations exist

for placing the service call to be subject to access control validation, and to have a command generated in the DIXI console. The actual auxiliary code generation is performed by a tool provided with DIXI.

When a service needs to invoke another service's call, the developer should keep in mind that the call done through the service call delegate is asynchronous. To obtain the result, the service class needs to implement a call-back method, decorated with the proper annotation and registered with the delegate call. Anonymous call-back interfaces can also be used, acting as listener classes similar to Java's event calls. In either case, a different method should be implemented to accommodate for the success outcome of the call as well as the failure condition.

The asynchronous nature of the service calls means that the service call implementation loses the method's internal state between the call-backs. DIXI helps bridge this gap by providing a context class. The framework ensures that the call-back receives the correct context, and by being able to receive any object within its storage, the internal state is preserved. Care should be taken, however, because the context instance is shared with every service thread presently handling the service call.

The client development basically involves importing and using static classes which are auto-generated stubs according to service's exposed interfaces. These stubs feature blocking until reception of the result. In addition to ordinary calls, it is possible to register a call-back listener interface implementation for being notified of events if services support them.

1 Introduction

Developing a distributed system presents a need for a way to place components that can appear in one or more locations, and connecting the components so that they can act in concert or complement each other. If the component hosting environment can also provide the capabilities of a message bus, both the runtime and the development can benefit.

For XtreemOS we are thus building a framework for developing, hosting and exploiting services that compose essential components of the XtreemOS system. We name this framework DIstributed XtreemOS Infrastructure (DIXI).

Communication for the services is an important feature of the framework, but not an exclusive one. In this respect we have introduced and described the DIXI framework in [1]. This current deliverable extends the previous entries by providing further information on the architecture, and contains the documentation needed by anyone who would like to use or work with the framework.

We start the deliverable by describing the motivation for developing DIXI. In Section 2 we describe the technical background of the framework and its architecture. The developer's guide, presented in Section 3, provides the information needed for writing or updating services hosted in DIXI, complete with example code. Developers wishing to write client side code that needs to exploit services in DIXI can also refer to this section. Users wanting to set up and run DIXI along with the hosted services can refer to Section 4. We conclude the deliverable with Section 5.

1.1 Motivation

In the early designing stages of the XtreemOS services it became clear that the services developed for the system would be experimental at first, but they would soon stabilise into production-level software. To this end, we needed a framework that would

- offer quick prototyping of the services,
- provide a staging environment for the XtreemOS services,
- simplify a development of a distributed system, composed of services,
- provide a communication layer between services running on the same node,
- provide communication between the nodes.

While the first services to use DIXI were those of AEM [2], the framework is general and can host any service package's components. We considered known

and widely used middleware products such as CORBA or web services. However, our middleware message bus and the hosted components needed to satisfy these requirements:

- The services should be as light-weight as possible.
- The target system would be highly distributed.
- The system should provide such a level of abstraction that the programmer would not have to implement communication-related logic. The framework would take care of the communication.
- The services would be written in Java.
- Peer services could use ready-made Java interfaces for issuing service calls.
- The client code could be written either in Java or in C.
- The communication layer would be interchangeable with any protocol required by the developers.

In this respect, frameworks such as CORBA require both the client and the server code to contain considerable amount of code for looking up services, negotiating their types and other CORBA-aware operations before it can actually use the service. Web services, on the other hand, require installation of servers like Apache Tomcat and Axis, introducing additional overhead in system confugiration, service and client programming, as well as the run-time overheads.

We therefore decided to develop DIXI, which we chose to implement using a staged event-driven architecture (SEDA) [3]. The outcome is a flexible framework that does not depend on external middleware instances. Instead, it uses ordinary TCP/IP for its communication, with an option to use SSL for encryption.

During the development we have performed the benchmarking, reported in [4].

1.2 Uses of DIXI

The high level of abstraction and generality of DIXI permits virtually limitless uses. In the XtreemOS context, DIXI provides the following capabilities:

- Staging of **services** that can be either ordinary stateless services or they can provide an intrinsic state. They act as core services, providing functionalities of the cluster or grid to the software packs, or exist as representatives of specific nodes.

- **Messaging and communication** between nodes, using plain TCP/IP, or, to provide security and trust, using SSL for encryption and authentication of the peers in the communication.
- Certain services can provide **bridging** the DIXI-hosted services with other components.
- Built-in client programs represent **commands and utilities** extending the services' capabilities towards the users. These components enable the usability of the system within DIXI.
- Further, the client-side libraries can be built into **web applications**, providing further types of front-end to the DIXI services.
- Custom **user clients** can take advantage of the system, e.g., the jobs submitted to the XtreemOS VOs.

As a result, the DIXI in XtreemOS hosts services of AEM [5], certain security infrastructure services such as VOPS and RCA [6], it contains services that wrap the functionality of the SRDS, and the front-ends take shape of the command-line utilities and the VOLife web interface. The framework is also fully extensible, as shown by the case of the introduction of the VNodes [7].

2 Architecture

In this section we describe the architecture of the part of the DIXI framework that is dedicated to the runtime of the system. In terms of deployment, it consists of the following components:

DIXI comprises the following components:

- The DIXI Daemon, also named simply the **XOSD**, usually runs as a Linux service on the host system. It, in turn, provides the staging environment of the DIXI services, their cooperation and communication.
- The client interface that can be imported into user programs. The **XATI** library provides the Java bindings, and the **XATICA** library provides the C bindings.

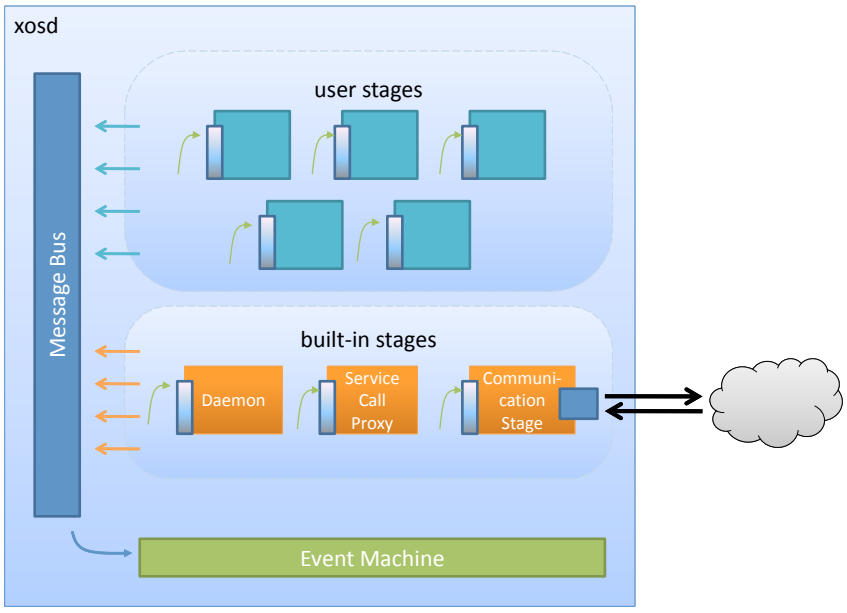

Figure 1: Architecture of the DIXI Daemon.

The xosd's architecture is depicted on Figure 1. It shows the following essential components:

- **Stages** are the classes which usually implement **services**. They run in one or more separate threads within the xosd's memory space and expose service

calls towards other services. Each stage has its own messaging queue for storing incoming service messages. Most of the stages are provided by the user and can be freely included or excluded within an xosd's runtime. However, certain stages are built in and represent an important part of the DIXI's features:

  - **Daemon** stage represents the book-keeping stage, holding the information of all the stages running within an xosd instance, providing their runtime manipulation, and the ability to control the xosd.
  - **Service Call Proxy** is a stage for acting on behalf of the clients connecting to the xosd.
  - **Service Call Redirector** stage handles the requests to the services that run in another instance of the xosd.
  - **Communication Stage** provides the capabilities of the network connectivity. It employs the Apache Mina[1] framework for handling TCP/IP and SSL connections with other xosd hosts.

- **Event Machine** is a root object, responsible for instantiating stages and connecting their messaging queues.

- **Message Bus Stage** is an object, which provides communication capabilities between various stages within an instance of xosd. The service messages use it to post their service messages. The Message Bus Stage collects the service messages in a FIFO manner and, based on the service header information, passes them to the recipient service's event handler.

The arrows in Figure 1 represent the passing of the service messages, which form the following cycle:

- A service message originates from one of the stages within the xosd. It contains a service call request, the target service and any parameters.

- The Message Bus collects the messages in an internal FIFO queue. The Event Machine collects the service message from the tail part of the queue and de-marshals sends them to the target stage.

- The target queue executes the service request, returning a result. This result is intercepted by DIXI, addressed to the stage originating the request. and queued again with the Message bus.

[1] http://mina.apache.org/

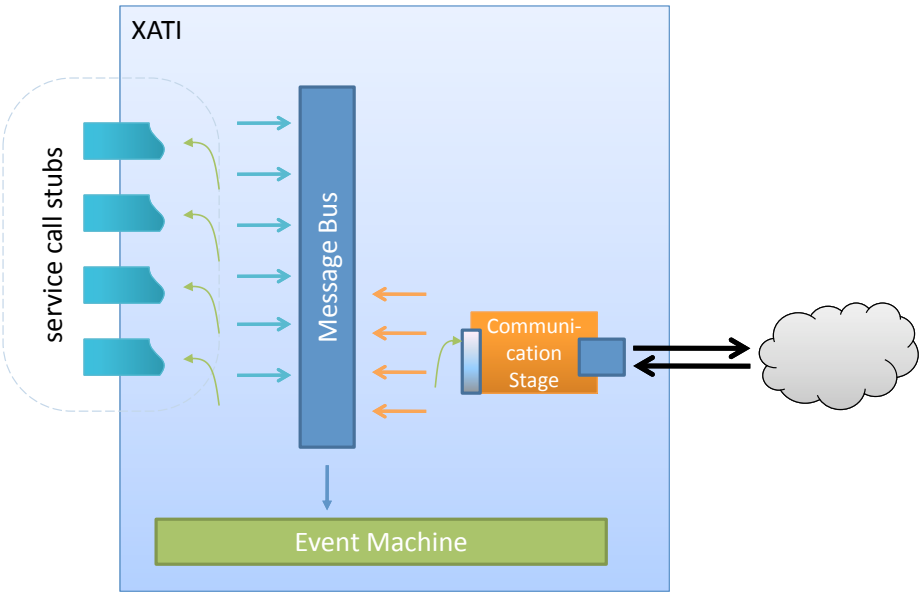

Figure 2: XATI architecture.

A special case occurs if the target stage does not exist within the xosd instance. In this case the Service Call Redirector will look it up from the service directory and re-queue the service message to be sent through network to the host of the service. All the network-bound messages are handled by the Communication Stage. The Communication Stage also acts as a network server, receiving service messages from remote clients and queuing them with the Message Bus.

In terms of their implementation, the services are decoupled from the rest of the DIXI layers. This is made possible by employing generated service call stubs (service call delegates), making certain that the stage implementation, when invoking other services, does not need to explicitly form and send the service messages. Instead, the delegates provide the means to perform asynchronous Java calls. Such calls do not receive the result of the call, but register a call-back that the DIXI will call to provide the result.

Any DIXI stage can be a client to another DIXI stage. However, to unlock the full potential of using the software deployed in DIXI, the user programs can use **XATI** or **XATICA** client libraries for invoking service calls. Figure 2 represents the architecture of the XATI library. The architecture of the XATICA is similar, except that it runs in C. As the figure shows, XATI uses a paradigm of message queueing similar to that in DIXI. However, the only stage that remains active is that of the Communication Stage, providing the connectivity with xosds.

In XATI, the service messages arrive in the Message Bus through generated service call stubs. These are designed to block the execution of the thread until

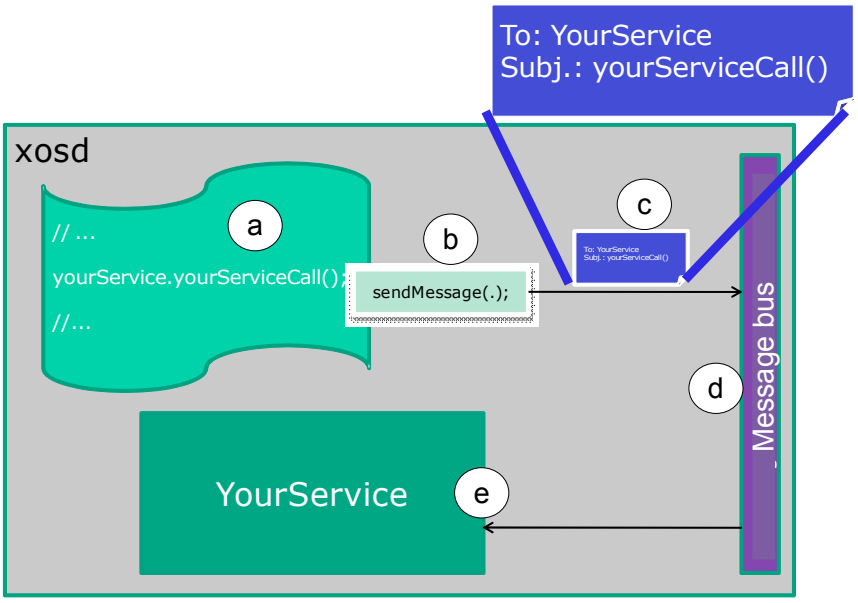

Figure 3: Service call invocation when the service is located within the same xosd.

the Event Machine receives the result to be returned to the calling thread.

Service messages and information flow. The payload of the communication between the stages has the form of service messages. These are implemented as Java classes, serialised and deserialised depending on the transport. Based on their needs, DIXI uses the following service message types:

- **ServiceMessage**: the basic class for queueing in the Message Bus, containing information such as the name of the target service, the name of the method to be invoked and the call parameters.
- **CallbackMessage**: similar to the **ServiceMessage**, but with the purpose to describe the return path for the message call result. It is usually included as a part of the ServiceMessage's payload, but can be also used stand-alone for the services that support notifications.
- **NetPackage**: a wrapper type for **ServiceMessage** used where the target service needs to be contacted through the network.

The service messages are thus exchanged within the DIXI based on the needs of the call. The Figure 3 shows the most basic scenario of invoking a service message of a service presently running within the same xosd instance. In this scenario,

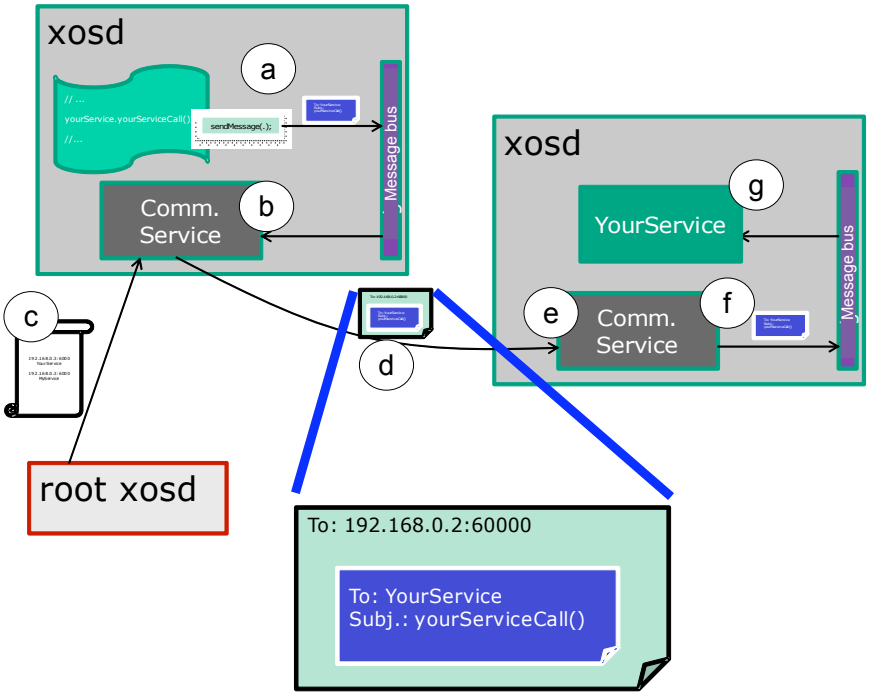

Figure 4: Service call invocation when the service is located on another node.

a service containing an instance of a delegate class calls the delegate method (a). The called stub forms a **ServiceMessage** (c) and sends it to the infrastructure (b). The Message Bus (d) receives it and, the Event Machine (not shown) executes it in the target service (e).

A common scenario involves invoking a service which resides on another instance of the xosd, shown on Figure 4. Like in the previous scenario, the client service invokes a delegate call which implicitly forms a **ServiceMessage** passed into the Message Bus (a). The Event Machine cannot invoke this message, thus it passes it to the service call redirection stage. This obtains a service directory (c) and forms a **NetPackage**, passing it to Communication Stage (b). The latter contacts the Communication Stage of the target host (e) and sends the service message over (d). The remote Communication Stage extracts the **ServiceMessage** and queues it with its local Message Bus, effectively having the target service invoked (g).

One advantage of such an architecture is that an xosd can receive on the same port or, effectively, through a single network channel the service invocations for any service hosted within. It is therefore also possible to set up an xosd that will route the service messages through a firewall or a NAT subnet.

3 Developer's guide

This section is directed at the developers who would like to create services to be staged in the DIXI framework. It describes the basic elements that make a Java class into a DIXI service, provides the directions on how to develop the asynchronous calls, and specifies the rules governing the DIXI services. For illustration, many section contain examples containing sections of working code or contents of other text files. The end of each example is marked with a small square □.

3.1 Service development

3.1.1 Overview

A service in DIXI is a class implemented in Java. It can contain an internal state, and it exposes a set of methods. The purpose od a service is to provide interrelated functionalities. For example, a phone directory service maintains a list of names and phone numbers as its internal state, and provides methods for adding, removing and listing the entries.

Implementation of a DIXI service is very similar to implementing a call that other classes use by calling its public methods from the same Java process. However, we need to have the following in mind:

1. The service calls cannot be invoked directly by calling the class implementation's method. Instead, a suitable access point akin to remote procedure calls needs to be invoked.

2. When calling another service, the call will not be blocking until the called method returns a value. Instead, a call-back needs to be provided that will receive the returned value.

The service process implementation occurs in two phases that can freely interchage:

1. The work on the source code that implements the service.

2. Using the `XServiceProcessor` for generating auxiliary code as well as service access points and XATI (API and access points for the clients). The tool uses reflection to extract the API

There are many ways of going about implementing a DIXI service:

- Starting from the scratch, we can implement the API as the DIXI service class, and put the implementation within the DIXI class. This is suitable for a quick development of prototype classes that are ready to be tested right away on a system that will use DIXI as the main message bus. However, this may lead to the code where all the processing happens within the DIXI class, and such code might be difficult to read.

- Alternatively, if there is a pre-existing library or a pre-developed code containing the implementation that we would like to use as a DIXI class, then we can create a DIXI service class which serves as a wrapper to this implementation.

For instance, in XtreemOS, services such as the `JobMng` and the `ReservationManager` implement the service within the DIXI class. On the other hand, `AllocationMng` or `SRDSMng` are wrappers for another class.

3.1.2 Requirements

The main requirement for being able to effectively develop DIXI classes is to use a platform which *supports Java 6* or later. The code generation tool takes advantage of the Java's decorations for extracting the relevant information from the code when auto-generating auxiliary classes. In this way, all interfaces and stubs emerge from the actual code itself, avoiding the need to prepare separate interface specification documents. Both Sun's Java[2] and the OpenJDK[3] are supported.

3.1.3 Preparing the service implementation class

In order for a class to become a service in DIXI, it needs the following:

- Annotate the class with `@XOSDSERVICE` and

- Extend the class by by `Abstract2wayStage`.

The following imports are required for the base class and the annotations:

```
import eu.xtreemos.annotations.XOSDSERVICE;
import eu.xtreemos.system.eventmachine.stage.Abstract2wayStage;
```

The declaration of the class then appears like in the following example:

[2]http://java.sun.com
[3]http://openjdk.java.net/

```
@XOSDSERVICE
public class MyService extends Abstract2wayStage {
   \\ ....
}
```

3.1.4 Initialising the class

The service class can maintain a state in the form of its class member field values. Any needed initialisation, such as instantiating classes, should be placed into the class's `void init()` overload. This method will be called by the xosd after instatiating the service handler classes.

To notify the xosd or any other classes that the initialisation has been successful, do one of the following within your implementation of the `void init()`:

- Before the end of the class, if the initialisation is successful, call `super.initialise()`.
- Alternatively, invoke the notification by calling the inherited `notifyServiceInitialised()` method.

If the initialisation contains critical code that may fail, the hosting *xosd* will usually not stop functioning, but rather proceed with hosting all the other services. Considering that the xosd runs as a daemon on the Linux system, the failure to initialise will go unnoticed, save for any error messages in the message log, if the implementation outputs them. This means that the next time a user will try to use the service's calls, the outcome will be undefined or it will throw a seemingly unrelated exception.

For handling the way the *xosd* handles the services with failed initialisation, it is possible to use the following inherited fields:

- `boolean isInitialised` — if set to `true`, the service will be considered initialised, and the service handler will invoke the service call methods as requested. Otherwise, it will allways throw a `ServiceNotRunningException` with the message from the value of `errorMessage`.
- `String errorMessage` — use this field to assign the error message to be contained in the service exception, if the service is not initialised properly.

The example on Figure 5 demonstrates the usage of the initialisation overload.

```
public void init() {
    // the default is to indicate failure
    this.isInitialised = false;

    try {
        // tricky initalisation which may fail
        // ...

        // safe area, we succeeded
        this.isInitialised = true;
        notifyServiceInitialised();
    } catch (Exception ex) {

        // log the failure

        // set the message to be shown with any
        // attempt to use the service
        this.errorMessage = "Cannot initialise " +
            "my service: " + ex;
    }
}
```

Figure 5: Initialisation of a service.

3.1.5 Writing service calls

All public methods of the service class become available to other services as service calls.

Additionally, the service calls can be annotated with the following annotations:

- `@XOSDXATI` marks a method to appear in the XATI API. The annotation has two optional parameters:
  - `String returntype()` — optional parameter assigning the return type of the call. This does not have to be set because the proper return type will be assigned to XATI API return type.
  - `boolean debug()` — optional parameter putting the method into the XATI API for debug purposes only.
- `@XOSDCONSOLE` — create an entry command for the XConsole which invokes this call via XATI. The annotation has the following parameters:

  - `String returnType()` — optional parameter assigning the return type of the call. This does not have to be set because the proper return type will be assigned to XConsole command's return type.
  - `String consoleString()` — the string representing the command that the XConsole will recognise to be the call to this method.

- `@XOSDCALLBACK` marks the method as a call-back, creating a corresponding helper method into the call-back class (see Section 3.1.6).

- `@XOSDCHECKVALIDITY` causes the xosd to validate the certificate by trusted signers and the date-based validity. The following parameters can be set:

  - `String certificate()` signifies the name of the parameter to be used for validation.
  - `boolean debug()` enables or disables the debugging mode for the validity checks.

- `@XOSACCESSCONTROL` enables the access control for the service call. The access control in DIXI is based on the categories that the methods can be assigned to at the design-time, and the access control list read at the run-time. The method needs to define a formal parameter of class `X509Certificate` to contain the caller's credential. The Section 3.1.8 contains a detailed description of the use of this annotation. Here we provide a brief summary of the parameters that can be set:

  - `String category()` defines the category of the method subject of the access control checks. A single category can be set, or, if needed, multiple comma-separated names of the categories can be provided.
  - `String credential()` defines the name of the parameter to be used as a credential. It has to be the name of one of the parameters defined in the method's signature.
  - `boolean verifyKey()` enables or disables the requirement to verify the credential against the private key.
  - `checkValidity()` enables or disables the requirement to also check validity of the credential. Setting the parameter to **true** is equivalent to decorating the method with `@XOSDCHECKVALIDITY`.

The **XServiceProcessor** uses the visibility and the decoration information to generate the stubs and the auxiliary code that is necessary for the infrastructure

to function properly. There are many uses of the XServiceProcessor, but in the following sections we will focus on the **service call stubs**, the **XATI interfaces** (i.e., the Java client interfaces), and the **call-back** helper classes.

The service call's parameters as well as the return values can be any serializable Java class. Please note that currently the primitive Java types are not supported.

XATICA-friendly parameters. If you are targeting your service also towards the C-based clients, you may need to provide the method signatures such that they will be compatible with the XATICA library. The Section 3.4.4 provides a table with the currently supported Java classes usable in XATICA.

It is also possible to extend the support to own custom classes. To do this, have your class that appears as a type of a parameter or a return value of a service call, implement the interface `IXATIParasable`. The interface should provide the implementation of the following methods:

- `IXATIParsable parseXATI(String toParse)` — this method should use the contents of the `toParse` parameter to instantiate an instance of the class.
- `String toXATIString()` — this method should describe the instance in such a way that it can be parsed into a clone of the instance using `parseXATI`.

Warning: when the string is transmitted to xosd, it attempts to parse it as an XML. Therefore if your string representation of the IXATIParsable contains an XML, please make sure to escape it in the serialization, and un-escape it when instantiating from a string.

3.1.6 Calling other services' calls

The most important auto-generated classes contain the delegates that other services can use to perform the service calls. These delegates reflect the signature of the original service call, with the only difference, that all of them are of type void.

The delegate class, by default, appears in the **XOS_Services/src** project source folder, the `eu.xtreemos.xosd.services` package, and it is named as the service class with a prefix `S`. For example, the service class named `MyService` will yield a service call delegate class named `SMyService`.

For each service call method of the service class, the delegate class contains three overloads of the service call's counterpart:

- The delegate containing the same sequence of parameters as they appear in the service call.

- The delegate containing the same sequence of parameters as they appear in the service call, with one additional parameter representing the success call-back service message.
- The delegate containing the same sequence of parameters as they appear in the service call, with two additional parameters. The first additional parameter represents the success call-back service message, and the second additional parameter represents the failure call-back service message.
- The delegate containing the same sequence of parameters as they appear in the service call, with an additional parameter representing an anonymous call-back listener interface implementation.

Their implementation composes a service message corresponding to the service call, and sends it to the message bus. The delegates do not wait for the message to be delivered, but return immediately instead. The overloads with additional call-back parameters embed the call-back service messages within the newly composed service message. This provides a way for the invoked service to return the result of the service.

Example. Let us assume we have implemented a service named `MyComputation`. One of `MyComputation`'s methods has the following signature:

```
public Integer myCall(String name) throws Exception;
```

The XServiceProcessor puts class `SMyComputation` into the `eu.xtreemos.xosd.services` package of the **XOS_Service** project. The class will contain the following methods:

```
public void myCall(String name) throws Exception;
public void myCall(String name,
   CallbackMessage callback) throws Exception;
public void myCall(String name,
   CallbackMessage callback,
   CallbackMessage exceptionCallback)
         throws Exception;
public void myCall(String name,
   IServiceCallback callback) throws Exception;
```

□

To obtain an instance of the delegate class, use the constructor which takes a `Context` instance as a parameter. The instance is available within service calls. Please refer to section 3.1.10 for more details on using the context.

The *call-backs* are a special type of service calls. The call-backs are never invoked explicitly by the remote service, but they are invoked implicitly by the infrastructure in order to pass the result of the service call. There are two kinds of call-back methods:

- The success call-back. It obtains the return value of the service call.
- The failure call-back. This call-back is invoked if the service call throws an exception.

As the generated delegates show, DIXI provides two ways of defining the call-backs: the call-backs defined and implemented as the stage's method, or anonymous call-backs that use a listener interface implementation.

Methods defined as call-backs. These call-backs are ordinary methods defined and implemented within the service's implementation class and decorated with `@XOSDCALLBACK` and should follow these guidelines:

- The success call-back should have a single parameter with the type equal to the return type of the service call being called.
- The failure call-back should have a single parameter typed as `java.lang.Exception`.
- The return type of either call-back should be the same as the return type of the method which registered the call-back.

Java does not enable using the method pointers directly in the delegate stub, so instead a method description that is an instance of the `CallbackMessage` class is needed. An instance of this class contains all the information needed by the DIXI to identify the host, service and the call that the result needs to be returned to. The **XServiceProcessor** code generator tool will use the signature of the `@XOSDCALLBACK` methods to create a call-back helper class with static methods that create and return the `CallbackMessage` descriptions.

By default, the call-back helper class is named like the service class, prefixed with letters `CB`, and it is placed into the `.service` sub-package of the package containing the service class. To obtain the service class description, call the static method named like the call-back method..

Example. Let us assume that, in addition to the `MyComputation` service class, we will have access to the service calls defined in the service named as `YourService`. The service exposes the following method that `MyComputation` needs to call:

```
public String yourServiceCall() throws Exception;
```

We would like to invoke this service call from `MyComputation`'s `myCall`.

In effect, this means that during the design time we need to import into our project the library containing the `SYourService` class definition. Considering we would like `MyComputation` to be notified of the outcome of the `yourServiceCall`, we first implement the call-back method to process the return value. The return value of `yourServiceCall` is of type `String`, and this defines the call-back's parameter type. And since `myCall`'s return value is of type `Integer`, the same return type will be used in the call-back.

```
@XOSDCALLBACK
public Integer successCallback(String result) {
    System.out.println("Result: " + result);
    Integer returnValue = 42;
    return returnValue;
}
```

If there is an exception in `yourServiceCall`, we need another call-back to handle the failure in the service call. The difference with the success call-back's signature is that its parameter is always typed as Exception:

```
@XOSDCALLBACK
public Integer failureCallback(Exception e){
    System.err.println("Exception! " + e);
    Integer returnValue = new Integer(-1);
    return returnValue;
}
```

Once the call-backs are defined, we need to use **XServiceProcessor** to generate the call-back helper class for us.

We now need to implement the actual making of the call:

```
    public Integer myCall(String name)
                throws Exception {
        // initialise the delegate class
        SYourService yourService =
            new SYourService(curContext.get());

        // making the service call
        yourService.yourServiceCall(
            CBMyService.successCallback(),
            CBMyService.failureCallback());

        /* finishing the class, but not returning
         * anything yet */
        return null;
    }
```

In this example, we first created an instance of the `SYourService` delegate class, using the getter of the call's context (`curContext.get()`). We then use the service call method delegate, registering the two call-backs, which effectively sends a service message to the message bus requesting the service call invocation. We finish by returning `null` (please refer to section 3.1.7 for further details on returning the values). □

The figure 6 illustrates the execution flow of the service methods. The vertical (blue) arrows represent the ordinary Java line execution, while the arrows between them (the red arrows) represent the service call invocations. In the case of `myCall`, the invocation is requested explicitly, and the execution of `yourServiceCall` may be started before `myCall` finishes. The code of `yourServiceCall` is prepared like any Java method, the processing of which ends either at a valid `return` directive, or when a message is thrown. Depending on the outcome, either the success call-back or the failure call-back is invoked.

Anonymous call-backs. A standard way in Java for registering call-back-like notifications is to employ listener interfaces with methods that a component calls for us whenever something interesting happens. In DIXI, `IServiceCallback` is such an interface, defined as follows:

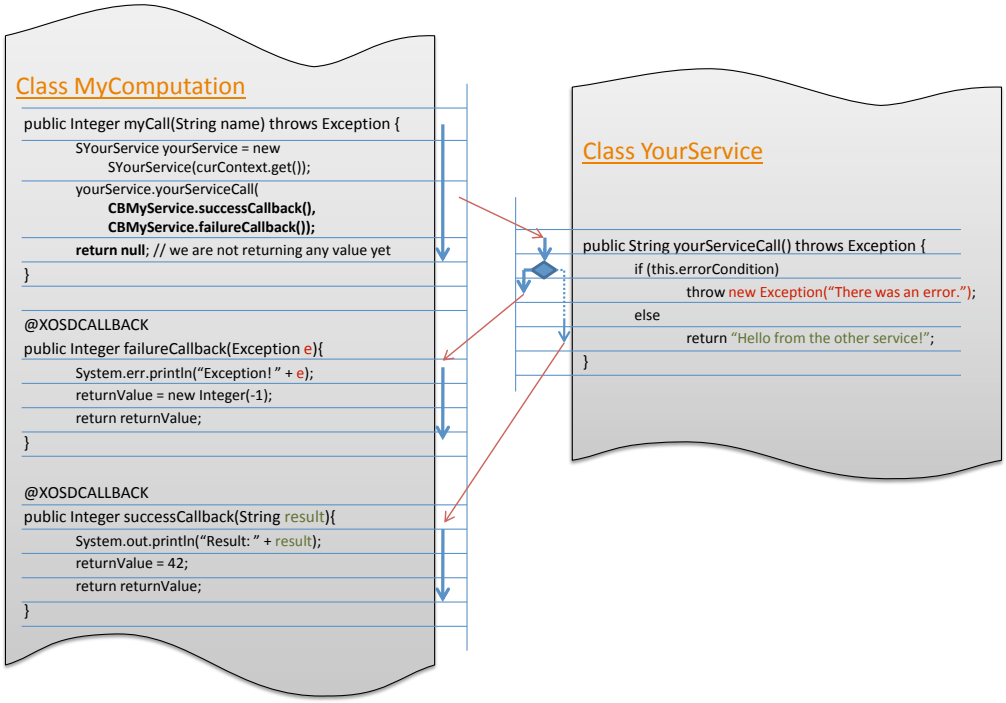

Figure 6: The call invocation sequence. The `myCall` requests the `yourServiceCall` to be invoked. In `yourServiceCall`, an error could occur, resulting in `failureCallback` invocation. Otherwise, `successCallback` is invoked with the return value.

```
public interface IServiceCallback {
    public Object success(Object result)
            throws Exception;
    public Object failure(Exception exception)
            throws Exception;
}
```

It therefore simply accommodates for both the success and the failure callback within one class. The advantage over the need to provide the methods decorated with `@XOSDCALLBACK` is that it requires no passing of the code through the code generation tool. Many developers will also prefer this method as it might produce a cleaner code. On the downside, however, the parameters and return values are general, so no strong type checking is possible at the design time.

```
public Integer myCall(String name) throws Exception {
    // initialise the delegate class
     SYourService yourService =
        new SYourService(curContext.get());

    // making the service call
    yourService.yourServiceCall(
        new IServiceCallback() {
            public Object failure(Exception exception)
                        throws Exception {

                System.err.println("Exception! " + e);
                returnValue = new Integer(-1);
                return returnValue;
            }
            public Object success(Object result)
                        throws Exception {
                System.out.println("Result: " + result);
                returnValue = 42;
                return returnValue;
            }
        });

    /* finishing the class, but not returning
     * anything yet */
    return null;
}
```

Figure 7: Using the anonymous call-back implementaion in a service call.

Example. The equivalent of the previous example using anonymous call-backs is shown on Figure 7.

Of course this example only implements the minimum required by the interface. Custom implementation classes can contain further elements, such as fields with call-specific information whenever needed in the call-back. □

3.1.7 Returning a value

When using Java class methods called directly and within the same memory spaces, each method returns a value only once, when it finishes the processing. Similarly, most of the services will expect that their registered call-backs will be called only once per each service call invocation. This also means that either the call will occur to either the success call-back or the failure call-back, never both.

In this section, we assume that the service call completes with no errors that would indicate a failure. Therefore, once it returns a value, the caller service will receive it through its success call-back.

In this fashion, an implementation of the service call needs to provide the return value only once, usually when it completes its processing.

Returning a value of a simple service call method (i.e., a method that does not make further service calls) is the same as returning a value in a normal Java fashion: by using the `return` keyword. The service handler will take care of sending the appropriate service message containing the returned value.

In a more complex scenario, the service call method makes service calls, and the processing of the fragmented into one or more call-back methods. In this case, the following rules should be observed:

- If the method calls other service calls and registers at least one call-back, the method **should return** `null`. This goes for the service call method or any of the subsequent call-backs.

- If the method is the final call-back of the computation flow, the return value is ready, and it can use it in its `return` directive.

- Returning `null` is valid. The registered success call-back of the callee service will be invoked after the last method in the call-back sequence of the called service returns.

If any of the call-backs in the invocation sequence of the service class return a non-null value, the callee service's call-back will receive it shortly after it is returned. However, the service's call-back sequence will resume in parallel, and any additional non-null return values will cause undefined behaviour.

3.1.8 Service call invocation access control

Services may implement methods that are sensitive in nature, and thus should be subject to access restrictions. Such methods can only be issued by the users that are capable of providing proper credentials. In XtreemOS such credentials are contained in the user's certificates as obtained from the VOLife or CDA server.

The service developers are free to build in the access control checks into their methods. A typical way to perform this is to include as a parameter a form of credential, then checking a credential attribute (e.g., the user's group and role) against an access control list. If the user's credential checks, then the method proceeds with its invocation. Otherwise, it aborts the operation with a failure.

There are several drawbacks with this process, however:

- The developer needs to take additional time to implement the access control checks, which could have been provided globally and uniformly. Custom access control checks also require an increase in the needed configuration files, particularly if each service uses its own scheme of access control.

- With the custom access control checks, the code execution already enters the implementation of the target service call, possibly posing a security threat.

- The service messages usually have no way to check the credential against the private key. This is due to the fact that the service receives a secondary call from another service, or the message arrives through the service proxy.

To solve this, the DIXI provides a built-in mechanism for performing the access control checks before invoking a service call. The methods are marked with *categories* to provide a simple way to maintain access control lists in the runtime.

In the design time, to have a method be checked for access control, decorate your methods with the `@XOSACCESSCONTROL` annotation.

Example. The following method, which is also available as a client call, will be available to the callers that are permitted to perform actions in the category `vo_administration`:

```
@XOSDXATI
@XOSACCESSCONTROL(category="vo_administration",
        credential="userCert")
public String getResourceList(String voID,
        X509Certificate userCert) {
 // ... implementation here ...
}
```

The name of the `credential` parameter of the decoration must match the name of the parameter used for access control check. In this example, the xosd will also perform the credential authenticity check, but will not perform any validity check. There is no restriction of the placement of the credential parameter.

To assign multiple categories for the call, use a comma-separated list of the category names:

```
@XOSACCESSCONTROL(category=
          "vo_administration, administration",
          credential="userCert")
public String removeVO(X509Certificate userCert,
          String voID) {
 // ... implementation here ...
}
```

□

Please note that whenever the decoration or any of its parameters change, you need to regenerate the code.

The access control list is a simple text file containing a list which, for each category, it defines which groups and roles are permitted to invoke the service calls assigned this category. The category name is delimited from the permission list with an equals sign =. Multiple permissions are delimited with a comma , and the group and role are expressed as `group:role`.

To express permission of any group and role, the asterisk $\star$ can be used.

The location of the access control list is defined in the **XOSdConfig.conf** in an option named as `accessControlListPath`. The following shows a valid access control list:

```
registration_info=*
resource_administration=admins:res_admin, admins:admin
administration=admins:admin
```

The access control is therefore based on the certificates which need to contain the proper values in the ”Group” (OID 1.34.5.0.14.7) and the ”Role” (OID 1.34.5.0.14.6) attribute. The user passes the XtreemOS user certificate as the credential, requesting an invocation of a service call. DIXI then uses the information on the method's categories to consult the access control list. If the user's group and role exist in the permissions list of any of the categories assigned to the service call, it will proceed with the invocation. Otherwise, it will throw the `PermissionDeniedException`[4] exception.

Credential verification. The credentials used in the service call access control are sensitive and should be subject to authenticity verification. Considering that the X509Certificate contain the public and freely exchangeable key, in principle anyone can obtain anyone else's credential and use it as a parameter value when

[4] eu.xtreemos.xosd.security.PermissionDeniedException

issuing a service request. To ensure that only the holder of the credential's private key counterpart issues the request, the system should verify it.

The DIXI uses the following mechanism to verify the certificate:

- The XATI client has to use SSL to connect to the xosd and issue the request. For the SSL, the same certificate (and corresponding private key) have to be used as the certificate passed in the credential parameter value. If mismatching certificates are used, the credential will not be verified and the server will return the `PermissionDeniedException`.
- The server that does not process the request, but only acts as the service call proxy for the client, will verify the credentials in the call parameters against the certificate used in the SSL communication. Any verified credentials in the call will be marked as authentic in an additional service call parameter.
- The target service which receives the client's service call request indirectly from the proxy will first check if the proxy is hosted on a trusted host. If so, it will base the access control decision on the additional information on the verification provided by the peer (i.e., the proxy).
- The service receiving a service call from another service (i.e., not from a XATI client) that is hosted on a trusted host will assume the credentials verified.

In summary, the credential verification relies on the SSL to provide the necessary verification of the credential against its private key. Further, it assumes that the trusted hosts will not use false credentials to gain access to restricted service calls. Currently, the trust of the peer service is based on the trusted issuer of the certificate used in the SSL handshake between the two services.

3.1.9 Throwing an exception

In Java, throwing an exception is a legitimate way of notifying of an error or a failure of any sort. In the DIXI services, it is also a valid way for the service to abort its processing whenever necessary. This includes any exception thrown within the service method implementation or the call-back method implementation, not surrounded in the `try` ... `catch` block.

It is a good practice to catch any exceptions that may occur in the service implementation's runtime by either calling one's own or other code. In this way the service implementation has an opportunity of cleaning up any side effects of the unfinished computation. If the caught exception renders the result meaningless, and if the caller has to be notified of the failure, the implementation has to throw an exception that is not caught by the user's code. This exception, however, will be caught by the xosd infrastructure, and invoke the caller's failure call-back.

XATICA-friendly exceptions. As explained in Section 3.4.4, the XATICA receives the outcome of the service call as an error code. To provide informative result in cases of failure, we recommend the use of the `XOSException` or its (possibly custom) extensions, that provide the means to assign the error code in the exception. The class provides one of the following constructors to be used or extended, where the `errCode` parameter will translate into the return value in the XATICA in case of a failure:

```
public XOSException(final int errCode);

public XOSException(final String message,
        final int errCode);

public XOSException(final String message,
        final Throwable cause, final int errCode);
```

3.1.10 Using the context

When a new service call is invoked, it creates a new *context*. The context consists of a state, representing this call only, that is isolated from other service call contexts. It contains the information on the manner in which the the call was invoked, the address of the calling service, as well as the information on the call-backs to be used to return the service's outcome. The context persists in all executed parts of the service call, fragmented into the main call and the call-backs.

The context is embodied in the instance of the class `Context`, located in the `eu.xtreemos.system.eventmachine.queue` package, that is obtainable using the `curContext.get()`.

A `Context` class instance has the following useful fields:

- **storage** — a field that can be used to store and obtain the call's state (i.e., the parameter and local variable values),
- **senderCertificate** — the peer certificate used by the remote host (the sender of the service message request) in the SSL session. This certificate always represents the other party in the point-to-point communication. If the value is `null`, the SSL was not used in the communication. The storage's value is preserved between the call-back methods.
- **callSenderAccessPoint** — the reported access point, i.e., the address of the xosd that sent the current service message. This address can be useful when the call-back needs to find out where the response arrived from. The value of this field will be unique for each method or call-back.

- **senderCommunicationAddress** — the address of the host that initiated the chain of the service calls.

Preserving the call state. Often, our service call needs to first obtain some information from another service, then, based on the result and the original set of parameters, computes the response and returns it. In practice, this means that the information available in the first method (the service call implementation) is required in the call-back method. The latter is out of Java scope of the first method. This means that all the local variable values that will be required in the subsequent call-back methods need to be stored.

The `Context`'s storage can receive any `Object`. If only a single variable needs to be preserved in the storage, the methods can readily assign and retrieve it, casting it into the proper type when referring to it. When multiple local variable values need to be transferred, however, it is best to prepare a storage class to hold the values that need to be retrieved in the call-back method.

Example. In this example, we will have a service which computes a price of a rectangular real-estate, using another service to obtain the price of a real estate unit.

We first define a class for storing the values in the parameters of the service call method. The class can be embedded into the service implementation class, or placed as a stand-alone class.

```
public class DimensionStorage {
    Double width;
    Double height;
}
```

Then we implement the service call method.

```
public Double rectangleArea(Double width,
        Double height) throws Exception {

    // obtain the current context
    Context context = curContext.get();

    // initialise the storage
    DimensionStorage storage = new DimensionStorage();

    // store the parameters
    storage.width = width;
    storage.height = height;
    context.storage = storage;

    // query the price
    SRealEstate estate = new SRealEstate(context);
    estate.queryPrice(
        CBMyService.queryPriceCallback());
}
```

Finally, we implement the call-back method which will use the values of the parameters.

```
@XOSDCALLBACK
public Double queryPriceCallback(Double price) {

    // obtain the current context
    Context context = curContext.get();

    // recover the storage values
    DimensionStorage storage =
        (DimensionStorage)context.storage;

    return storage.width * storage.height * price;
}
```

□

Scattering requests. A core service is a service which runs in one (or a small number) of instances within a site. These services usually have a number of node-level services, i.e., the kind of services which need to run on nodes of a specific kind. The core nodes may need to gather the responses of these node-level ser-

vices. To achieve that, it first scatters the requests, sending one to each node.

However, one needs to keep in mind that the requests will go out almost concurrently. The responses will then arrive from each node, and the call-back methods (either from success or failure) will be invoked in some order, which *will not be the same as the order we made requests in*. Both the success and the failure method should therefore

- keep count of the number of responses arrived so far. If the counter reaches zero or registers the same number of invocations as the number of called nodes, it should return the final result, otherwise it should return `null`. Please note that each call can result either in the success or failure call-back, so each call-back should account for the call. Also

- use context's `callSenderAccessPoint` to check the node that sent the particular response. If the member field's value is `null`, the call arrived from the same xosd.

Example. The following code scatters the requests to services on nodes contained in the collection `nodeList`:

```
Context context = curContext.getContext();

// ...

// Initialise the invocation counter
context.storage = nodeList.size();

for (CommunicationAddress node : nodeList) {
    // make the next request be sent to node
    yourService.setRemoteAddress(node);
    // a targeted service call
    yourService.yourServiceCall(successCB,
        failureCB);
}
```

The context's storage in this example holds the invocation counter initialised by the number of invocations. Each of the two call-backs decreases the counter until it reaches zero. The counter should be accessed in a thread-safe way unless your service will always run in a single thread.

```
    @XOSDCALLBACK
    public Integer successCB(Integer result) {
        boolean finish = false;
        Context context = curContext.getContext();
        synchronized (context.storage) {
            context.storage = (Integer)context.storage - 1;
            finish = (context.storage == 0)
        }

        if (context.callSenderAccessPoint == null) {
            // the response was sent from this xosd
        } else {
            // the response was sent from another xosd
        }

        if (!finish)
            return null; // we are not done yet
        else
            return result; // we are done
    }

    @XOSDCALLBACK
    public Integer failureCB(Exception ex) {
        boolean finish = false;
        Context context = curContext.getContext();
        synchronized (context.storage) {
            context.storage = (Integer)context.storage - 1;
            finish = (context.storage == 0)
        }

        if (context.callSenderAccessPoint == null) {
            // the exception arrived from this xosd
        } else {
            // the exception was sent from another xosd
        }

        if (!finish)
            return null; // we are not done yet
        else
            return result; // we are done
    }
```

□

Scenario	Effect in DIXI
Undirected service call. Service look-up fails.	`ServiceNotRunningException` to caller's failure call-back
Undirected service call. Target service off-line.	Remove target address from local list. Retry with the next on the list.
Undirected service call. All known targets off-line.	Obtain a new list and retry.
Undirected service call. No known targets.	`ServiceNotRunningException` to caller's failure call-back
Directed service call. Target service off-line.	`MessageSendingException` to caller's failure call-back

Table 1: Service message invocation failures and respective outcomes.

3.1.11 Network call invocation problems

Certain service call invocations will invariably require a network communication to be carried out. This introduces possibilities that the communication will fail. The table 1 summarises situations covered by DIXI and possible outcomes. They all involve a client service residing on a different node from the target service. Upon the call, communication bus consults a local copy of service directory and selects for target the first node on the list. If one does not exist yet, it tries to get it from the global service directory.

3.1.12 Custom call-backs and notifications

Some of the services that we are implementing may need to provide a way to notify other services or the clients about specific event occurrences. For other services, it is possible to notify them by calling their exposed service messages. The problem arises when we need to send notifications to the clients. The clients, unlike the services, are not discoverable and behave differently.

The solution is to let the client register for receiving the call-backs by providing the API call which receives an `ICallback` object. It is up to the service implementation then to store the call-back object and use it whenever notification is due. It should also provide a way for the client to cancel the reception of the notifications.

To invoke a call-back, the `SendCallback` method is available in the `Abstract2wayStage` class, and thus to all the service class implementations:

- `public void SendCallback(ICallback callbackToSend, Object result, ICallback failureCallback` — request a call-back invocation. The `callbackToSend` parameter should contain the

call-back description registered by the client. The `result` parameter will become the value of the call-back method's parameter. The `failureCallback` parameter can be the description of the *service's* call-back to be invoked if the invocation of the call-back fails (e.g., due to the network failure). Can be `null` if no such call-back is needed.

This method can be used from another service call, call-back, or, e.g., from a thread. Please note, however, we can pass two `ICallback` parameters to this method. The `callbackToSend` parameter will typically have a value obtained from the client in another service call. The `failureCallback`, however, is a standard failure DIXI call-back with `Exception` for its single parameter type. The `CB`-prefixed auto-generated class methods can therefore be used here.

Please refer to section 3.4.3 for the information on how to implement the client for it to take advantage of the call-back functionality.

3.2 Configuration classes

The DIXI framework and its development tools feature a way to write and read configuration files that reflect the Java classes containing the configuration in their field member. It includes reading and writing of many standard Java types, such as ArrayList<String>, String[], String, CommunicationAddress, int, long, double.

The configure file takes shape as a properties file. This means that each line in the configuration file contains the name of the property, followed by the equals sign and the string representation of the field's value. The following example of XOSdConfig.conf shows the attributes representing the address, integer, and an array of strings:

```
#Properties File for the client application
#Thu Oct 18 09:15:19 CEST 2007
rootaddress.host=/192.168.0.246
rootaddress.port=60000
services.size=10
services.8=eu.xtreemos.service.test.service.TestHandler
services.6=eu.xtreemos.xosd.jobDirectory.service.JobDirectoryHandler
services.5=eu.xtreemos.xosd.jobmng.service.JobMngHandler
services.3=eu.xtreemos.xosd.resourcemonitor.service.ResourceMonitorHandler
services.2=eu.xtreemos.xosd.xmlextractor.service.XMLExtractorHandler
services.1=eu.xtreemos.xosd.resmng.service.ResMngHandler
services.0=eu.xtreemos.xosd.daemon.DaemonGlobal
```

To use a properties file for persistent storage of configuration values, we can create a class (e.g. MyClass with some fields), annotate it with @XOSDCONFIG annotation and extend it with eu.xtreemos.generator.config.IGenerateConfig (part of XOS_SEDA). Implement the defaultConfig() method where the initialisation of the fields' default values takes place. (If a config file is not provided, these values

will be used when the user creates an instance of this config class.) If you do not add implementation of the IGenerateConfig interface, the generated configuration class will implement defaultConfig() (so that user can add initialisation code later).

```
@XOSDCONFIG
public class XOSdConfig implements IGenerateConfig{

    public CommunicationAddress rootaddress =
      new CommunicationAddress();
    public String[] services = new String[10];

    public void defaultConfig() {
        services[0] = "eu.xtreemos.xosd.daemon." +
                "DaemonGlobal";
        services[1] = "eu.xtreemos.xosd.resmng." +
                "service.ResMngHandler";
        services[2] = "eu.xtreemos.xosd.xmlextractor." +
                "service.XMLExtractorHandler";
        services[5] = JobMngHandler.class.getName();
        services[6] = JobDirectoryHandler.class.getName();
        services[8] = "eu.xtreemos.service.test." +
                "service.TestHandler";
        rootaddress.host =
                CommunicationAddress.localNetAddress();
        rootaddress.port = 60000;
    }

}
```

Run the XServiceProcessor generates, for example, the class CXOSdConfig (with prefix “C”, in the same package as XOSdConfig), which extends XOSd-Config and enables the user to load and save field values from/into provided config file.

There are two methods of using the generated class. One can use it statically, if the same values should be used throughout the project:

```
/ filename is path to property file
/ Loading
XOSdConfig.load2Instance(filename, xosdconfig);
/ Saving
osdconfig.rootaddress.port = 60000;
osdconfig.services[0] = "newService";
XOSdConfig.saveInstance("./newxOSDCONFIG.conf",
    xosdconfig);
```

Alternatively, one can create an instance of the configuration class to contain the values in the configuration file:

```
    // filename is path to property file
    // create an instance with values stored in the file
    public static CXOSdConfig xosdconfig =
        new CXOSdConfig(filename);
    // store values into new property file, which
    // can be loaded later
    xosdconfig.save(myNewFilename);
```

3.3 Using the code generation tool (XServiceProcessor)

The `XServiceProcessor` is the tool that analyses the code, compiles it internally, then uses Java reflection to extract the relevant information from the classes.

To define which classes to analyse, the tool uses the service_folders.txt file as an input of the paths. The following shows an example of the contents of the file:

```
../XMLExtractor/src/
../JobDirectory/src/
../RCAServer/src/
../XOSd/src/
#END
```

It is possible to exclude any line by putting the hash (#) character to the beginning of the line.

In Eclipse, one can use the tool by importing the XServiceProcessor project into the workspace, then run the `Runner` Java class. For the utility to function properly, the runtime needs to reference all the classes compiled in the target projects as well as any external library referenced by the service implementation classes.

The service code generation is possible from the command line as well. The

XServiceProcessor project contains the `xserviceprocessor.sh` which can be called to invoke the tool. Before calling it, the service_folders.txt has to contain the list of paths to the sources containing the code to be processed. Further, make sure that the target source is compiled before invoking the tool. It is best if the compilation produces one or more jars that will appear in the tool's class-path.

The script supports one of the following uses:

```
# ./xserviceprocessor.sh \
        -ClassesBase ~/xtreemos/grid/dixi-aem/trunk/
```

With the `-ClassesBase` option the tool will use for its classpath the classes and jars contained within the project pointed to in the path. The output files will be written into the XATI, XATICA and XOS_Services folders of the `dixi-main/trunk/` module. This is currently the most common usage.

```
# ./xserviceprocessor.sh \
        -ModuleBase ~/xtreemos/grid/dixi-aem/trunk/
```

The `-ModuleBase` usage differs from the `-ClassesBase` in the output files ending in the XATI, XATICA and XOS_Services folders contained within the **target** module (i.e., `dixi-aem/tunk/` in the case of the example).

```
# ./xserviceprocessor.sh \
        -AllInOne ~/xtreemos/grid/srds/trunk/
```

The `-AllInOne` option tells the tool to place all the output files inside the target project. It there creates or uses no separate projects (XATI, XATICA or XOS_Services).

3.4 Client development

3.4.1 Overview

In the terminology we use in the document, the client of the DIXI infrastructure is a program that itself does not run any DIXI services. The administrator and the user commands packaged in the XtreemOS Linux are all clients. They all take advantage of the communication library and the exposed interfaces provided by XATI.

3.4.2 Using XATI interfaces

The services can expose a subset of their service calls to the clients by using the `@XOSDXATI` decoration (section 3.1.5). The XServiceProcessor creates a class containing the methods with their signatures transcribed from the service class. These methods are static, and can be readily called to make the service call invocation from the client. Unlike the service call delegate class methods, described in 3.1.6, these calls *block until receiving the result*. This means that the service calls in XATI can be used like any other static class methods.

By default, the XServiceProcessor places the generated client API class into the XATI folder, into the `eu.xtreemos.xati.API` package, using the name of the service class prefixed with the letter `X` for the class name.

Example. Let us assume, like in the previous examples, our service is implemented in the `MyComputation` class, exposing the following method:

```
@XOSDXATI
public Integer myCall(String name) throws Exception;
```

In order for the client to make the call to this method, it needs to first import the generated XATI class:

```
import eu.xtreemos.xati.API.XMyComputation;
```

From within the code, wherever the call needs to be made, it uses the static method:

```
// ...

try {
    Integer result = XMyComputation.myCall("Test call");
    System.out.println("The call returned " + result);
} catch (Exception ex) {
    System.err.println("There was an error: " + ex);
}
```

As we can see, the client programmer does not need to register any call-backs. The return result and any exceptions thrown within the service call will be received by the client's code as if it were calling a local class method. □

3.4.3 Client call-backs

Some services provide the notification functionality, effectively pushing a message or a value to the client. The client needs to implement the handler of these notifications to be able to receive them. It then needs to first register the handler with the XATI, and, finally, with the service.

Implementing the call-back listener. The client needs the code that handles the call-back whenever invoked by the service. The DIXI provides the `ICallbackListener` in the `eu.xtreemos.xosd.basic` package with the following method to be implemented:

- `public void callbackInvoked(Object data)` — the method that will be called whenever a notification occurs. The `data` parameter will receive the value of the notification.

Registering the call-back listener with XATI. The XATI library will be responsible for calling the call-back listener when the appropriate message arrives. As a result, it needs to issue the call-back message description that the client will use in the last step, the registering the call-back with the service. This process requires a set of interfaces that may be confusing. The table 2 clarifies them.

Interface	Purpose	Needed by
`ICallbackListener`	Implements the handler	Client, XATI
`ICallback`	Describes service message	Service, XATI

Table 2: The interfaces involved in the client call-backs.

To assign the call-back implementation and to obtain the appropriate description, use the following static method of XATI:

- `public static ICallback setCallback(ICallbackListener callbackListener)` — register the call-back listener implementation with XATI and obtain the call-back message description related to the call-back listener.
- `public static void removeCallback(ICallback callback)` — unregister the call-back and stop receiving the notifications. This method expects the call-back message description.
- `public static void removeCallback(ICallbackListener callbackListener)` — unregister the call-back and stop receiving the notifications. This method expects the call-back listener implementation, and effectively stops any notifications arriving to this implementation.

Registering the call-back with the service. This final step involves notifying the service that the client needs to receive notifications. The specific call depends on the service and its API. If you are implementing your own services and would like to support call-backs, please refer to section 3.1.12.

Typically, however, the service needs to receive the call-back message description that contains the information identifying the client (e.g., the address), as well as the information needed by XATI to relate the service message with its respective call-back listener.

Example. On a XATI client, we might want to be notified about an event on the server. We thus first create a call-back invocation class instance:

```
ICallbackListener myListener = new ICallbackListener() {
    public void callbackInvoked(Object data) {
        System.out.println("Invocation with " + data);
    }
};
```

We then register it to XATI, obtaining the call-back message description. This ensures that XATI will call our listener implementation whenever the proper message call arrives.

```
ICallback myCallbackMsg = XATI.setCallback(myListener);
```

We now register the call-back message with the service, e.g., the `EventProducer` service.

```
XEventProducer.registerClientCallback(myCallbackMsg);
```

Of course, the `EventProducer` service has to have a service call exposed to XATI. On the service side we therefore need to first add a collection to hold all the clients' call-back descriptions, and implement the registration service.

```
@XOSDSERVICE
public class EventProducer extends Abstract2wayStage {

    // ...

    protected ArrayList<ICallback> callbacks;

    // ...

    @XOSDXATI
    public void registerClientCallback(ICallback callback)
    {
        this.callbacks.add(callback);
    }

    // ...

}
```

Finally, we implement another method in `EventProducer` that the service will call whenever the event of interest will occur, and thus notify all the registered clients by sending them the call-back message. Along with it we will also implement an optional call-back that will be invoked every time a client call-back invocation fails.

```
    protected void notifyClients(Object eventResult) {
        for (ICallback cb : this.callbacks) {
            SendCallback(cb, eventResult,
                CBEventProducer.callbackFailed());
        }
    }

    @XOSDCALLBACK
    public void callbackFailed(Exception ex) {
        // ... handle the failure here
    }
```

□

3.4.4 Using XATICA libraries

The client programs to DIXI services can also be developed in C. The standard `XATICA` directory of the DIXI repository contains the libXATICA library and the header files as well as the source needed to compile the library[5].

To use the XATICA library, your code needs to include the `XATICATypes.h` to have the XATICA utilities and structures available. The code generation tool will also create a header file named after the service class name, prefixed with `XC`. The service calls are translated into the global functions with the modification of its signature as follows:

- The return value of the method is appended to the parameter list. This return-value parameter is passed by reference of a type equivalent to the Java method's return type.
- The service calls with void return type produce no additional parameter to the signature.
- The return value of the function is the error code of the call.

Example. An equivalent to the XATI example in 3.4.2 in C would be as follows: □

Type mapping. The DIXI services can use any standard or user-defined Java classes for the service call parameters and return values, and the XATI client, being in Java, can use them directly. For the C clients, however, XATICA supports only a selection of the classes in Java that can be used by the C clients.

The Table 3 shows the mappings between the supported Java classes that appear in DIXI service calls. To gain access to the custom XATICA-related types such as `CommunicationAddress`, the source needs to import the `XATICATypes.h`.

The DIXI service implementations may provide further support to the classes used by the services. As explained in Section 3.1.5, the Java classes implementing `IXATIParsable` can instantiate from a string. On the XATICA side, all `IXATIParsable` implementations appear as `const char *`.

[5]Please note that libxml2-dev and libssl-dev are required to be able to compile the XATICA library.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <XATICATypes.h>
#include <XCMyComputation.h>

#define SOCK_MAX_BUF 4096

int main(int argc, char *argv[])
{

    // ...

    int returnValue;
    int retCode = myCall("Test call", &returnValue);

    if (retCode != 0)
        fprintf(stderr, "There was an error: %x\n",
            retCode);

    // ...
}
```

Figure 8: A sample program using XATICA.

4 User's guide

4.1 Service deployment

The DIXI services are not stand-alone programs. Instead, they are stages that need to be hosted within the XtreemOS Daemon xosd. The xosd, when started, will read a configuration file and load, in turn, the **service handler** of every stage that it finds listed in the configuration.

The xosd instantiates the service handler dynamically, based on the fully qualified class name (i.e., the string that contains both the package name and the class name of the handler). In order for the instantiation to succeed, the target handler class needs to be within the Java class path.

The xosd designates one or more threads[6]. It creates access points by listening to communication ports for incoming service messages. The incoming service messages get queued and passed on to the target service handlers. The overall

[6]By default, a single thread is assigned to a service, but more can be configured to run.

Java class	C equivalent
`String`	`const char *`
`Integer`	`int`
`Boolean`	`char`
`X509Certificate`	`const char` (PEM-formatted)
`CommunicationAddress`	`const typedef CommunicationAddre`
`MetricsDesc`	`const typedef MetricsDesc`
`MetricEvent`	`const typedef MetricEvent`
`ArrayList<String>`	`const typedef ArrayList`
`ArrayList<CommunicationAddress>`	`const typedef ArrayListCA`

Table 3: Supported Java classes mapped into C for XATICA.

operation of the xosd depends on the settings in its configuration files.

4.2 The xosd command-line parameters

The xosd is a program that can be run with the following syntax:

```
xosd [-C config_path] [-c xosd_config_file] [-r root_dir]
     [-s server_ip[:port_num] ]
```

- **-C** *config_path* sets the folder to contain all the configuration files used by the services. If the option is omitted, then the **/etc/xos/config/** folder is used. Overrides the **-c** directive for the xosd's configuration.
- **-c** *xosd_config_file* sets the path and file name to be used for xosd configuration. If the option is omitted, **/etc/xos/config/XOSdConfig.conf** is used instead. *Please note that if the value denotes a relative path, the daemon and the services will use config_path as the absolute path prefix.*
- **-r** *dir_path* instructs the daemon to set the DIXI root folder to the value of *dir_path*. By default this is the install path **/usr/share/dixi**.
- **-s** *server_ip[:port_num]* sets the ip address *server_ip* and, optionally, the port to *port_num*, of the *root xosd*. This directive overrides the related settings in the configuration files.

4.3 Running xosd as a service

In the XtreemOS Linux distributions, the xosd installs with scripts that allow it to be used like a standard system service.

The user can use **/etc/init.d/xosd** script for running, stopping and restarting xosd. This script supports the commands *start*, *stop* and *restart* at this time.

```
root# /etc/init.d/xosd {start|stop|restart}
```

On the Mandriva distributions, it is possible to use the **service** command instead of calling the script directly. By default, the logging information is placed into **/var/log/xosd/xosd.log**.

4.4 Configuring the xosd

The xosd reads the contents of a text file containing the parameters for its operations. As explained in 4.2, the default configuration file's full path is **/etc/xos/config/XOSdConfig.conf**, but the xosd can be instructed to look for the configuration files in another location using the `-C` option, or use a different path and file altogether, supplied with the `-c` option.

The configuration options can be grouped into logical groups, as explained below.

Defining which services to run. Listing the services to run within the xosd involves specifying the service's fully qualified class name and, optionally, the number of threads (if more than 1) to start for the service. There are two mutually exclusive ways of enumerating the services:

- Providing a list of services in the **XOSdConfig.conf**. This involves the **services.size** and the **services.***X* options.
- Defining the service's invocation by placing **.stage** files into a specific folder. This involves the **xosdStagesSubDirectory** option in the **XOSdConfig.conf**. If used, the list of services in the **XOSdConfig.conf** will be completely ignored.

To provide the list of services in the **XOSdConfig.conf**, edit as needed the following options:

- **services.size** — the number of services to be run by this xosd. This option defines the size of the list of services to be read by xosd.
- **services.***X* — one or more entries listing the services to be run by this xosd, with $0 \leq X <$ **services.size**. Make sure each service has a distinct *X* suffix, otherwise only the last entry with the same number will be used. Omitting entries is not an error, but also note that entries with indexes higher than the defined array size will be ignored with no error reported.

```
# Enumerate the services to be run on the node
services.size=7
# This service needs to be included for
# proper operation of xosd
services.0=eu.xtreemos.xosd.daemon.DaemonGlobal
services.1=eu.xtreemos.xosd.security.rca.client.service.RCAClientHandler
services.3=eu.xtreemos.xosd.resourcemonitor.service.ResourceMonitorHandler
services.4=eu.xtreemos.xosd.execMng.service.ExecMngHandler
services.6=eu.xtreemos.xosd.xmlextractor.service.XMLExtractorHandler
```

Figure 9: Listing the services in the **XOSdConfig.conf**.

The Figure 9 shows a sample portion of the **XOSdConfig.conf** listing the services. Notice that the **services.2** and **services.5** entries are missing, which is a valid setup (i.e., the services can be omitted from the start-up by deleting or commenting out the lines defining the service).

For manipulating the services' inclusion in the xosd using scripts and helper programs, we have developed an alternative way to define which services to run. In the **XOSdConfig.conf** we set the following option:

- **xosdStagesSubDirectory** — the directory to contain **.stage** files which define what services need to run in the xosd. If the directory is relative, the xosd will assume it is a subdirectory of the configuration path (please refer to the `-C` command-line option in 4.2). By default, the **xosd_stage** value is used, meaning that the files are expected to be in **/etc/xos/config/xosd_stages/**.

The xosd will look for the files with names ending with **.stage** in the designated directory. The files may be empty, or contain certain options. Depending on each of the file's name and contents, one service will be run per file. The **.stage** files can contain the following options:

- **service** — the name of the service to be loaded. If the option is omitted, the file's name without the **.stage** suffix will be used.
- **numThreads** — the number of threads to start the service in. The default value is 1.
- **enabled** — can take values **true** (default value) or **false**. This option lets the user to disable loading of a certain service even if the respective **.stage** file is present.

For example, we put the following line into the **XOSdConfig.conf**:

```
xosdStagesSubDirectory=xosd_stages
```

We prepared a number of **.stage** files in this directory:

```
$ ls -l /etc/xos/config/xosd_stages
-rw-r--r-- 1 [...]  133 [...] DaemonGlobal.stage
-rw-r--r-- 1 [...]    0 [...] eu.xtreemos.xosd.security.
rca.server.service.RCAServerHandler.stage
-rw-r--r-- 1 [...]  158 [...] RCAClientHandler.stage
```

This listing suggests the xosd will load three stages. The RCAServerHandler's stage file is empty, therefore the **eu.xtreemos.xosd.security.rca.server.service.RCAServerHandler** will be used as the class name for the service to load. DameonGlobal.stage contains the following:

```
$ cat /etc/xos/config/xosd_stages/DaemonGlobal.stage
service=eu.xtreemos.xosd.daemon.DaemonGlobal
enabled=true
numThreads=1
```

This means that the **eu.xtreemos.xosd.daemon.DaemonGlobal** will be the class name used when instantiating the service. Finally,

```
$ cat /etc/xos/config/xosd_stages/RCAClientHandler.stage
service=eu.xtreemos.xosd.security.rca.client.service.RCAClientHandler
enabled=false
numThreads=5
```

the RCAClientHandler would not be loaded, because the content of the file tells the xosd that the service is disabled.

Defining the paths and files important for xosd and services.

- **xosdRootDir** — the string containing the path to the xosd root. This path is used by some of the services to locate files needed for their proper operation.
- **accessControlListPath** — the path to the file containing the access control list. Please refer to Section 3.1.8 for further details.

Communication parameters. The xosd provides stages that provide connectivity to the xosds and clients elsewhere on the network. The following parameters modify these values.

- **useSSL** — indicates whether the communication should use SSL for security. Default value: *false*.

- **trustStoreSSL** — indicates the path to the folder containing the signer certificates trusted in the SSL handshakes.

- **trustStore** — indicates the path to the folder containing the signer certificates trusted in the AEM or other services.

- **certificateLocation** — the path to the public certificate used for the SSL handshakes. The certificate needs to be able to handle both server and client connections.

- **privateKeyLocation** — the path to the private key used for the SSL handshakes and for encrypting the communication.

- **networkInterface** — an optional option that tells the xosd which network interface to use for the inbound traffic, e.g., `eth0` or `eth1`. If **externalAddress**, **rootaddress.externalAddress** and **rootaddress.host** are left out from configuration (are commented out), **networkInterface** is used to set up IP proprely assuming XOSd root is running on this node.

- **xosdport** — the port the xosd will use for listening to the inbound traffic.

- **externalAddress** — optional parameter. It defines the IP address or the host name of the gateway xosd that nodes from other subnets can use to connect to this node. If this node is running behind a firewall, then the router or firewall that responds on the external address IP needs to forward the inbound traffic arriving to port **xosdport** to this node. If the local subnet has multiple DIXI nodes, then one of the nodes will act as proxy, and the the firewall needs to forward the traffic on the port number as configured in that node's **XOSdConfig.conf**'s **xosdport** value. If the parameter is omitted, the same address is used as that of the xosd's host.

- **xmlport** — the port used for the inbound XML connections from C applications using XATICA C library.

- **rootaddress.host** — the IP address or the host name of the **root xosd**. Can be the node's own address for a stand-alone or a root xosd. This address needs to be the one the root xosd's host's local address, as seen from the peers in the host's subnet. If the host is running on a different subnet than this node, the rootaddress.externalAddress has to be set to the address exposed to the current node.

- **rootaddress.port** — an optional parameter defining the port number the **root xosd** is listening to for requests. Default value is 60000.

- **rootaddress.externalAddress** — the IP address or the host name of the node visible from external network nodes. The node with this IP needs to have the port **rootaddress.port** forwarded to one of the nodes that on the internal network run the xosd.

Figure 10 shows an example **XOSdConfig.conf** which configures the local xosd to connect to a *root xosd* running at the address myroot.xlab.si. The communication will be SSL-encrypted, using the SSL credentials of the local node. The daemon will start up and serve the services defined by the **.stage** files in **/etc/xos/config/xosd_stages**.

```
# This is a sample configuration.
# The lines starting with the hash # are ignored.

# Look for the .stage files here
xosdStagesSubDirectory=/etc/xos/config/xosd_stages

# Security and SSL-related settings
trustStore=/etc/xos/truststore/certs/aem_trusted/
useSSL=true
trustStoreSSL=/etc/xos/truststore/certs/
certificateLocation=/etc/xos/truststore/certs/resource.crt
privateKeyLocation=/etc/xos/truststore/private/resource.key

# Node's parameters and root addresses
xosdRootDir=.
networkInterface=eth0
xosdport=60000
xmlport=55000
externalAddress=testnode2.xlab.si
rootaddress.host=myroot.xlab.si
rootaddress.externalAddress=gateway.xlab.si
rootaddress.port=60000
```

Figure 10: A sample DIXI daemon configuration file.

When the xosd starts, the hosted services become available at a common port of the selected or configured network interface. The Java services and clients would use the port 60000 to send their service messages, and the C clients would use the port 55000 to exchange the XML-encoded service messages.

The services use the **communication address** structure when addressing their service messages. The communication address is composed of the target host address, the port number, and, optionally, the subnet's gateway (external address).

For example, `192.168.0.30:60000(194.249.173.87)` represents a service running on an xosd that uses host address 192.168.0.30, accepts the incoming messages at port 60000, and is running on a subnet which has a gateway xosd running on host with address 194.249.173.87.

4.4.1 Connecting the nodes

The system becomes distributed when the services communicate with their peers on other nodes. Starting up the xosd is the first step, which makes the services available for the rest of the subnet. The next step is to advertise the service's access point (i.e., the address of the xosd) to the other services. This is important for the type of services that occurs only on one node in the whole system, or uses a number of replicated mirrors. Hard-coding or using configuration files would be inconvenient and quite unscalable. Therefore, the infrastructure takes care of keeping the service locations, and enables the service access point look-up. The redirection mechanism relies on this look-up to pass an undirected service message to a target service.

Currently, the DIXI supports a static and centralised node and service directory. This means that we have to designate a node and its xosd to become a *root xosd*. This xosd will keep a list of other nodes, and also store the list of services as reported by the other xosds.

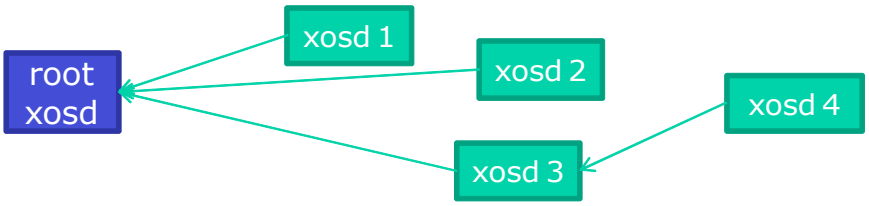

Figure 11: Connecting the xosds to the root xosd.

All the other xosds in the system need to specify the address of the root xosd in their configuration file (please also refer to Section 4.4). The Figure 11 shows a possible set-up of registering xosds with the root xosd. Here, the root xosd started up first, followed by the other xosd as numbered on the figure. The xosd 4 would connect to xosd 3, which would pass on the information to the root xosd.

Please note that the xosds can reside within the same node or on multiple nodes. Furthermore, the chart shown on Figure 11 does not represent an overlay of message passing, as the message transport happens on a point-to-point basis. Other xosds can communicate to the ones shown even if they are not connected to the same root xosd, as long as they know the address.

In the upcoming releases, however, a support for the SRDS will be added to replace the centralised solution with the distributed one.

Connecting the clients. Clients use XATI or XATICA (Java or C, respectively) to access the services from user programs and prepared commands. The XATI library is, in effect, a slightly repackaged /bf xosd which runs only the basic communication stages. It does not contain any lists of services or their addresses, and therefore it needs to connect to an active xosd in order for the service call invocations to function properly.

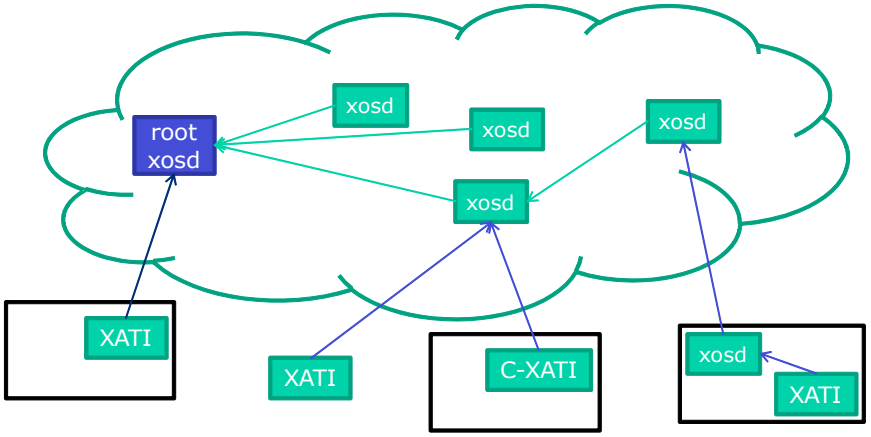

Figure 12: Connecting the client programs to the DIXI system.

Figure 12 illustrate how the clients can connect to an xosd. The clients can be within the network that is running XtreemOS, or even on a node that runs an active xosd. It can also connect to an xosd from an external node (such as a laptop or a home computer), but provided that the target xosd has an externally visible address, and no firewall blocks the access point's port. It is also possible to start an xosd on the external computer which reports its presence to the XtreemOS network, and then have the clients connect to this local xosd.

5 Conclusion

In this deliverable we presented the details on the implementation and the usage of the Distributed XtreemOS Infrastructure (DIXI). The framework has proven perfect for writing a new distributed operating system. With the built-in utilities, it reduces the effort needed to prepare the service interfaces, and with the architecture it enables a high level of separation between the service implementation and the nature of the underlying transport. The benefit is two-fold: the service developers can focus on the functionality of the services, while it is possible to add framework-related features without having to alter the service implementation in any way.

XtreemOS system provides a reference implementation and usage of the DIXI. The framework, however, can be used stand-alone or in any other system. The fact that it is developed in Java also enables it to connect any number of platforms into its grid. Deployed on the grids such as the PlanetLab[7] or the Grid5000[8], it offers a baseline for experiments needed in the distributed system research.

As part of the future work, certain improvements in terms of the usability will be implemented. Mainly these belong to adding diagnostics and management tools. It will particularly improve the usability by adding graphic user interfaces for monitoring and managing the run-time of the deployed DIXI framework, either as a Web application or a desktop GUI.

References

[1] XtreemOS Consortium. Basic service for resource selection, allocation and monitoring. Deliverable D3.3.4, November 2007.

[2] XtreemOS Consortium. Requirements and specification of XtreemOS services for application execution management. Deliverable D3.3.1, November 2006.

[3] Matt Welsh. SEDA: an architecture for highly concurrent server applications. http://www.eecs.harvard.edu/∼mdw/proj/seda.

[4] XtreemOS Consortium. Evaluation Report. Deliverable D4.2.6, November 2009.

[5] XtreemOS Consortium. Basic services for application submission, control and checkpointing. Deliverable D3.3.3, November 2007.

[7] http://www.planet-lab.org/

[8] http://www.grid5000.fr/

[6] XtreemOS Consortium. Fourth Specification, Design and Architecture of the Security and VO Management Services. Deliverable D3.5.13, December 2009.

[7] XtreemOS Consortium. AEM Prototype — Advanced features. Deliverable D3.3.7, March 2010.