Project no. IST-033576

XtreemOS

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

Second set of engine extensions covering XtreemOS functionality from D3.1.5
D3.1.9

Due date of deliverable: November 30^{th}, 2009
Actual submission date: December 4^{th}, 2009

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

Type: Deliverable
WP number: WP3.1
Task number: T3.1.2

Responsible institution: VUA
Editor & and editor's address: Thilo Kielmann
Vrije Universiteit
Dept. of Computer Science
De Boelelaan 1083
1081HV Amsterdam
The Netherlands

Version 1.0 / Last edited by Thilo Kielmann / December 4, 2009

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	$\surd$
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
0.99	23/11/09	Mathijs den Burger, Thilo Kielmann	VUA	complete draft version
1.0	004/12/09	Mathijs den Burger, Thilo Kielmann	VUA	final version, after internal reviewing

Reviewers:
Thorsten Schütt (ZIB), Marjan Sterk (XLAB)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.1.2	A runtime engine for dynamic call dispatching	VUA*

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

Executive Summary

This report describes the status of the implementations of the XOSAGA API engine extensions, according to the API extensions specified in deliverable D3.1.5 [10]. The implementations are available for the Java and Python engines of XOSAGA. Implementations within the C++ engine will be ready for the final XOSAGA API engine implementation, due with deliverable D3.1.11 [13]. Unfortunately, the XOSAGA `stream` package (the interface to WP3.2's *Distributed Servers*) turned out not to be implementable due to a severe granularity mismatch for connection hand-off operations. It has been replaced by a new `ds` package, specified in the appendix to this deliverable. Its implementation will also come with D3.1.11 [13].

Contents

1 Introduction **3**

2 XOSAGA Sharing Package **3**
2.1 Scalaris and XOSAGA 4
2.1.1 Java implementation 4
2.1.2 Python implementation 7

3 XOSAGA Distributed Servers Package **7**

4 XtreemFS URL Scheme **8**

5 Summary and Ongoing Work **8**

A Revised API Specification for Distributed Servers **10**
A.1 Specification Details 11
A.2 Example 19

1 Introduction

This report describes the status of the implementations of the XOSAGA API engine extensions, according to the API extensions specified in deliverable D3.1.5 [10]. It follows previous reports on the design of the XOSAGA API family (D3.1.1 [4] and D3.1.2 [7]), and their implementations (D3.1.3 [5], D3.1.6 [9], and D3.1.8 [12]). As such, this report focuses on the API extensions from D3.1.5, rather than describing the complete XOSAGA API family. The final deliverables D3.1.10 and D3.1.11 will provide such comprehensive descriptions of the XOSAGA API and its implementations.

The implementations of the API extensions from D3.1.5 that have been built so far, and that are described in this report, are built into the Java engine for XOSAGA, following the majority of XtreemOS services that provide Java interfaces as their major language support. As our Python implementation of XOSAGA is layered on top of the Java engine, support also has been implemented for Python. Implementations within the C++ engine will be ready for the final XOSAGA API engine implementation, due with deliverable D3.1.11.

Unfortunately, the XOSAGA `stream` package (the interface to the *distributed servers* from WP3.2) turned out not to be implementable due to a severe granularity mismatch for connection hand-off operations. As such, it has been replaced by a new `ds` package, as specified in the appendix to this deliverable. Its implementation, along with the C++ implementation of the `sharing` package, will come with D3.1.11.

2 XOSAGA Sharing Package

The following table summarizes the classes and interfaces of the XOSAGA `sharing` package, the respective XtreemOS backend services providing the underlying functionality, the programming languages in which these will be supported, and by which deliverable. Please note that it is not possible to support the OSS-related classes in either Java or Python, as these rely on memory-mapped regions that are only supported in C/C++.

class/interface	backend	C++	Java	Python
shared_buffer_service	OSS	D3.1.11	n/a	n/a
consistency_domain	OSS	D3.1.11	n/a	n/a
transactional_consistency_domain	OSS	D3.1.11	n/a	n/a
weak_consistency_domain	OSS	D3.1.11	n/a	n/a
shared_buffer	OSS	D3.1.11	n/a	n/a
shared_buffer_id	OSS	D3.1.11	n/a	n/a
callback	Scalaris	D3.1.11	D3.1.9	D3.1.9
shared_events	Scalaris	D3.1.11	D3.1.9	D3.1.9
shared_properties	Scalaris	D3.1.11	D3.1.9	D3.1.9

2.1 Scalaris and XOSAGA

Scalaris is a publish-subscribe ring on top of a scalable, transactional, distributed key-value store. The XOSAGA `sharing` package exposes the public-subscribe ring as `SharedEvents` objects, and the key-value store as `SharedProperties` objects.

`SharedEvents` allow XOSAGA applications to publish events under certain topics. Both events and topics are strings. Applications can subscribe to certain topics for which they want notifications, and also unsubscribe from them.

`SharedProperties` provide a distributed key-value storage. Applications can put, get and remove the key-value pairs.

Both `SharedEvents` and `SharedProperties` expect a bootstrap URL that specifies the location of the Scalaris server. For example, the bootstrap URL would be '`boot@localhost`' when the Scalaris bootstrap server is running locally.

We have created two implementations of the XOSAGA API to Scalaris defined in D3.1.5 [10]: one for Java, and one for Python. The C++ implementation is under development, due with D3.1.11, exploiting the experiences gathered with the Java implementation. Since Scalaris itself only provides a Java API, the XOSAGA C++ implementation will have to interface to the Java implementation via JNI.

2.1.1 Java implementation

The development source tree of the XOSAGA Java implementation can be found in the XtreemOS Subversion repository:

```
svn://scm.gforge.inria.fr/svn/xtreemos/grid/xosaga/java/trunk/
```

The API to Scalaris is part of the XOSAGA `sharing` package located in the subdirectory '`src/eu/xtreemos/xosaga/sharing`'.

The source code of Java XOSAGA can be compiled using Ant. Simply type 'ant' in the root directory of the development tree to compile all code into several JAR files. All JAR files (included all dependencies) will then be placed in a new subdirectory 'lib'. Javadoc of the XOSAGA API can be created by executing 'ant javadoc' in the root directory of the development tree. This will create a subdirectory 'javadoc' that contains all documentation in browsable HTML files.

The SAGA user guide explains how to develop and run a SAGA application. It can be found in the subdirectory 'doc'. In short, you must add all JAR files in the 'lib' subdirectory to your Java classpath. In addition, the environment variable 'saga.location' must contain the installation directory of Java XOSAGA (i.e. the directory that contains the lib subdirectory with all the JAR files). For convenience, the script 'bin/run-saga-app' performs these steps automatically, and requires only a single environment variable JAVA_SAGA_LOCATION to indicate the installation directory of Java XOSAGA.

XtreemOS already contains a compiled version of Java XOSAGA in the package xosaga-java, which can be installed via URPMI:

```
# urpmi xosaga-java
```

All Java XOSAGA files will be installed in /usr/share/xosaga-java. The script /usr/bin/run-saga-app will use this location automatically.

The implementation of the XOSAGA sharing package is built on top of the Java API provided by Scalaris. The API allows only a single connection to a Scalaris node. The 'Scalaris' object provided by the API is therefore wrapped into a singleton, so multiple instances of the SharedProperties and SharedEvents XOSAGA objects automatically reuse the single connection. Similarly, all SharedEvents objects internally share a lightweight HTTP server to receive updated values for subscribed topics. A user only has to provide a callback method that is invoked for each received update.

The following code example demonstrates the basic usage of the SharedEvents object. The example connects to a local Scalaris node, subscribes to the topic "mytopic", publishes a string in the topic and waits until it has received one new value for the subscribed topic.

```
import org.ogf.saga.error.SagaException;
import org.ogf.saga.url.URL;
import org.ogf.saga.url.URLFactory;
import eu.xtreemos.xosaga.sharing.Callback;
import eu.xtreemos.xosaga.sharing.SharedEvents;
import eu.xtreemos.xosaga.sharing.SharingFactory;
```

```
public class SharedEventsExample implements Callback {

    public static void main(String[] args) {
        new SharedEventsExample().run();
    }

    public void run() {
        try {
            URL u = URLFactory.createURL("boot@localhost");
            SharedEvents se = SharingFactory.createSharedEvents(u);

            String topic = "mytopic";
            System.out.println("Subscribing to " + topic);
            se.subscribe(topic, this);

            System.out.println("Publishing new value in " + topic);
            se.publish(topic, "Hello world!");

            // wait until new value has been received
            synchronized(this) {
                try {
                    wait();
                } catch (InterruptedException ignored) {
                    // ignore
                }
            }

            System.out.println("Unsubscribing from " + topic);
            se.unsubscribe(t);

        } catch (SagaException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }

    public void cb(SharedEvents se, String topic, String value) {
        System.out.println("Received notification in '" + topic
                + "': '" + value + "'");
        synchronized(this) {
            notifyAll();
        }
    }

}
```

2.1.2 Python implementation

The development source tree of the XOSAGA Python implementation can be found in the XtreemOS Subversion repository:

```
svn://scm.gforge.inria.fr/svn/xtreemos/grid/xosaga/python/trunk/
```

The source code of the `sharing` package described in deliverable D3.1.5 [10] is located in the file '`xosaga/sharing.py`'. Browsable API documentation generated by Epydoc is available in the subdirectory '`apidoc/html/`'.

The implementation is a thin layer on top of the Java implementation of the `sharing` package. Similar to the other Python XOSAGA classes, Python calls are forwarded to delegate Java objects that perform the actual work.

The implementation requires Jython as the Python interpreter. The script '`bin/jysaga`' can be used to start Jython plus Java XOSAGA with the right properties and classpath. Similar to the '`run-saga-app`' script in Java XOSAGA, the `jysaga` script uses the environment variable '`JAVA_SAGA_LOCATION`' to locate the Java XOSAGA installation directory.

3 XOSAGA Distributed Servers Package

The distributed servers, as implemented by WP3.2, provide a TCP stream interface to their clients. They achieve high availability and fault tolerance through forming a redundant group of server machines that can hand-over client connections to each other, without the clients noticing.

Distributed Servers provide location transparent networked services [3]. Clients connect to a single *distributed server address* for a service and may be moved transparently among multiple locations. Mobile IPv6 (MIPv6) route optimization [2] does the heavy lifting: all IPv6 connections from a client are atomically changed directly to each location, avoiding triangular routing. The distributed server address is simply an IPv6 [1] address. In the terminology of Distributed servers, a client first connects to a *contact node*. A client may then be transparently *handed off*—the server endpoint of all of the client's connections are transferred—to different servers for load-balancing or for client-specific processing. Distributed servers are described in Deliverables D3.2.2, D3.2.6 and D3.2.11 [6, 8, 11].

Deliverable D3.1.5 has described an XOSAGA `stream` package for Distributed Servers that had been designed in collaboration with WP3.2. This API package extends the SAGA `stream` package, by providing a stream class with an additional method *handover*.

In the course of the implementation work of the XOSAGA packages from D3.1.5, it has turned out, unfortunately, that an implementation of the XOSAGA `stream` is infeasible. This is due to the fact that that the distributed server implementation is working on the granularity of clients rather than individual streams. As a consequence, it had to be decided to not implement the XOSAGA `stream` package.

While this situation is causing a delay for the delivery of an XOSAGA API to the Distributed Servers, it also proves the strength of the overall API design process chosen for XtreemOS. The carefully packaged API design allows us to address the current issue without impacting any other part of the overall API. To resolve the issue, we have designed a new XOSAGA package for distributed servers in collaboration with WP3.2, called the `ds` package. This new API package is documented in the Appendix to this document. It will be implemented along with D3.1.11.

4 XtreemFS URL Scheme

D3.1.5 also specifies a proper URL scheme for addressing XtreemFS file system volumes, as e.g. `xtreemfs://users@xtreemfs.cs.vu.nl/home`. (This was an omission in earlier XOSAGA specifications that covered XtreemFS.) All existing XOSAGA implementations (for C++, Java, and Python) now use this URL scheme. No separate implementations are necessary.

5 Summary and Ongoing Work

This report describes the status of the implementations of the XOSAGA API engine extensions, according to the API extensions specified in deliverable D3.1.5. The implementations built so far, and described in this report, are built into the Java engine for XOSAGA, following the majority of XtreemOS services that provide Java interfaces as their major language support. As our Python implementation of XOSAGA is layered on top of the Java engine, support also has been implemented for Python. Implementations within the C++ engine will be ready for the final XOSAGA API engine implementation, due with deliverable D3.1.11.

Unfortunately, the XOSAGA `stream` package turned out not to be implementable. As such, it has been replaced by a new `ds` package, as specified in the appendix to this deliverable. Its implementation, along with the C++ implementation of the `sharing` package will come with D3.1.11.

References

[1] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6). RFC 2460, December 1998.

[2] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775, June 2004.

[3] Michał Szymaniak, Guillaume Pierre, Mariana Simons-Nikolova, and Maarten van Steen. Enabling service adaptability with versatile anycast. *Concurrency and Computation: Practice and Experience*, 19(13):1837–1863, September 2007. `http://www.globule.org/publi/ESAVA_ccpe2007.html`.

[4] XtreemOS Consortium. First Draft Specification of Programming Interfaces. Deliverable D3.1.1, November 2006.

[5] XtreemOS Consortium. First Prototype of XtreemOS Runtime Engine. Deliverable D3.1.3, November 2007.

[6] XtreemOS Consortium. First Prototype Version of Ad Hoc Distributed Servers. Deliverable D3.2.2, November 2007.

[7] XtreemOS Consortium. Second Draft Specification of Programming Interfaces. Deliverable D3.1.2, November 2007.

[8] XtreemOS Consortium. Reproducible evaluation of distributed servers. Deliverable D3.2.6, December 2008.

[9] XtreemOS Consortium. Second Prototype of XtreemOS Runtime Engine. Deliverable D3.1.6, November 2008.

[10] XtreemOS Consortium. Third draft specification of programming interfaces. Deliverable D3.1.5, November 2008.

[11] XtreemOS Consortium. Extended version of the distributed servers platform. Deliverable D3.2.11, December 2009.

[12] XtreemOS Consortium. Third Prototype of XtreemOS Runtime Engine. Deliverable D3.1.8, May 2009.

[13] XtreemOS Consortium. Final XOSAGA API engine implementation. Deliverable D3.1.11, March 2010.

Appendix

A Revised API Specification for Distributed Servers

```
package xosaga.ds
{
  class ds_service
  {
    CONSTRUCTOR     (in    saga::session        s,
                     in    string               name,
                     in    handoff_policy       policy = NULL,
                     out   ds_service           obj);

    DESTRUCTOR      (in    ds_service           obj);

    serve           (in    float                timeout = -1.0,
                     out   saga::stream         stream);

    get_client      (in    saga::stream         stream,
                     out   ds_client            client);

    get_all_clients (out   array<ds_client>     clients);

    get_all_streams (out   array<saga::stream>  streams);

    get_all_targets (out   array<saga::url>     targets);

    handoff         (in    ds_client            client,
                     in    float                timeout = -1.0,
                     out   saga::url            target);

    handoff_to      (in    saga::url            target,
                     in    ds_client            client,
                     in    float                timeout = -1.0);

    receive_handoff (in    float                timeout = -1.0
                     out   ds_client            client);

    close           (in    bool                 binding_reset = True);
  }

  class ds_client
  {
    CONSTRUCTOR     (out   ds_client            client);

    DESTRUCTOR      (in    ds_client            obj);
```

```
    get_url          (out   saga::url             obj_url);

    get_streams      (out   array<saga::stream>   streams);

    set_message      (in    saga::buffer          msg = NULL);

    get_message      (out   saga::buffer          msg);
  }

  interface handoff_policy
  {
    get_target       (in    ds_client             client,
                      in    array<saga::url>      options,
                      out   saga::url             target)
  }

  class round_robin_handoff_policy: implements handoff_policy
  {
    // no additional methods
  }

}
```

A.1 Specification Details

Class `ds_service`

The `ds_service` accepts new incoming connections as SAGA streams, allows to hand off all streams connected with the same client to another Distributed Servers node, and accepts such a handoff operation.

```
- CONSTRUCTOR
  Purpose:  create a service to access a running Distributed
            Servers daemon
  Format:   CONSTRUCTOR  (in   saga::session   s,
                          in   string          name,
                          in   handoff_policy  policy,
                          out  ds_service      obj);

  Inputs:   s:            session to be used for object creation
            name:         a name recognized by the local Distributed Server
                          daemon that maps to a local address
            policy:       the handoff policy to use
  InOuts:   -
  Outputs:  obj:          the newly created service
  PreCond:  -
  PostCond: obj can now serve incoming client connections.
```

```
  Perms:    -
  Throws:   IncorrectState
            IncorrectURL
            NotImplemented
            BadParameter
            NoSuccess
  Notes:    - if there is no local Distributed Servers daemon running, a
              'NoSuccess' exception MUST be thrown.
            - if the local daemon does not know the given name, a
              'DoesNotExist' exception MUST be thrown.
            - the local daemon MAY only allow one instance of a ds_service
              per name per machine. In that case, all subsequently created
              instances with a name that has already been used MUST throw an
              'AlreadyExists' exception.
            - the handoff policy can be NULL; in that case, the
              handoff() method without a target URL parameter
              will always throw an 'IncorrectState' exception.

- DESTRUCTOR
  Purpose:  destructor of the ds_service object
  Format:   DESTRUCTOR   (in   ds_service  obj)
  Inputs:   obj:          the ds_service object to destroy
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the service is closed
  Perms:    -
  Throws:   -
  Notes:    - if the service was not closed before, the
              destructor performs a close() on the instance,
              and all notes to close() apply.

- serve
  Purpose:  Wait for an incoming client connection
  Format:   serve          (in   float         timeout,
                            out  saga::stream  stream);
  Inputs:   timeout:        number of seconds to wait
  InOuts:   -
  Outputs:  stream:         new connected stream object
  PreCond:  -
  PostCond: - all postconditions of saga::stream_service.serve()
              apply.
            - the session of the returned stream is that of the
              ds_service object.
            - the associated ds_client object also contains the
              new stream.
  Perms:    - all permissions of saga::stream_service.serve() apply
  Throws:   NotImplemented
            BadParameter
```

```
            PermissionDenied
            AuthorizationFailed
            AuthenticationFailed
            IncorrectState
            Timeout
            NoSuccess
  Notes:    - all notes from saga::stream_service.serve() apply.

- get_client
  Purpose:  returns the client associated with a stream
  Format:   get_client     (in   saga::stream    stream,
                            out  ds_client         client);
  Inputs:   stream:         a connected stream
  InOuts:   -
  Outputs:  client:         the ds_client object associated with
                            the given stream
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   DoesNotExist
            BadParameterException
            IncorrectState
            NotImplemented
  Notes:    - if no client is associated with the given stream, a
              'DoesNotExist' exception MUST be thrown.
            - if the given stream is not connected, a
              'BadParameter' exception MUST be thrown.

- get_all_clients
  Purpose:  returns all clients currently handled by this
            ds_service object.
  Format:   get_all_clients  (out  array<ds_client>  clients);
  Inputs:   -
  InOuts:   -
  Outputs:  clients:          all clients currently handled by
                              this ds_service object.
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   IncorrectState
            NotImplemented
  Notes:    -

- get_all_streams
  Purpose:  returns all streams of all clients.
  Format:   get_all_streams (out  array<saga::stream>  streams);
  Inputs:   -
  InOuts:   -
  Outputs:  streams:         all streams of all clients
```

```
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
  Notes:    - the array is a shallow copy; streams served later
              are not reflected in the array.

- get_all_targets
  Purpose:  get the URLs of all the handoff targets.
  Format:   get_all_targets (out  array<saga::url>  targets);
  Inputs:   -
  InOuts:   -
  Outputs:  targets:  all possible handoff targets.
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
            PermissionDenied
            AuthorizationFailed
            AuthenticationFailed
            IncorrectState
            NoSuccess
  Notes:    - the number of possible handoff targets CAN be zero.

- handoff
  Purpose:  hand off all streams of a client to another node
            determined by the handoff policy of this ds_service.
  Format:   handoff      (in   ds_client  client,
                          in   float      timeout,
                          out  saga::url  target);
  Inputs:   client:       the client to hand off
            timeout:      number of seconds to wait
  InOuts:   -
  Outputs:  target:       the URL of the node selected for the
                          handoff
  PreCond:  - if an application-specific 'message' object has
              been set via ds_client.set_message(), it will be
              passed to the other server along with the handoff.
  PostCond: - the client object does not contain any streams anymore.
  Perms:    -
  Throws:   NotImplemented
            PermissionDenied
            AuthorizationFailed
            AuthenticationFailed
            BadParameter
            DoesNotExist
            Timeout
            IncorrectState
            NoSuccess
```

```
  Notes:    - the handoff target is determined by calling the
              get_target() method of the handoff policy of this
              ds_service
            - any exception thrown by the handoff policy MUST
              be forwarded
            - if the given client is not handled by this
              ds_service, a 'DoesNotExist' exception MUST be
              thrown.
            - if no handoff policy was given in the CONSTRUCTOR,
              an 'IncorrectState' exception MUST be thrown.
            - after a successful handoff, all subsequent method
              calls on the client MUST throw an 'IncorrectState'
              exception (except for the DESTRUCTOR and close()).
            - which streams are contained in the given client object
              is irrelevant; ALL streams from this client known
              by the local daemon are handed off.

- handoff
  Purpose:  hand off all streams of a client to a specific node
  Format:   handoff      (in   saga::url   target,
                          in   ds_client   client,
                          in   float       timeout);
  Inputs:   target:       the target server to which the client
                          has to be handed off
            client:       the client that is to be handed off
            timeout:      number of seconds to wait
  InOuts:   -
  Outputs:  -
  PreCond:  - if an application-specific 'message' object has
              been set via ds_client.set_message(), it will be
              passed to the target server along with the handoff.
  PostCond: - the client object does not contain any streams anymore.
  Perms:    -
  Throws:   NotImplemented
            PermissionDenied
            AuthorizationFailed
            AuthenticationFailed
            BadParameter
            DoesNotExist
            Timeout
            IncorrectState
            NoSuccess
  Notes:    - if the target does not exist, a 'DoesNotExist'
              exception MUST be thrown
            - if the given client is not handled anymore by this
              ds_service, a 'DoesNotExist' exception MUST be
              thrown.
            - after a successful handoff, all subsequent method
              calls on the client MUST throw an 'IncorrectState'
```

```
              exception (except for the DESTRUCTOR and close()).
            - which streams are contained in the given client object
              is irrelevant; ALL streams from this client known
              by the local daemon are handed off.

- receive_handoff
  Purpose:  receive all streams of a client that are handed off
            by another node.
  Format:   receive_handoff (in   float      timeout,
                             out  ds_client  client);
  Inputs:   timeout:         number of second to wait
  InOuts:   -
  Outputs:  client:          the client object that has been
                             handed off
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
            IncorrectState
            PermissionDenied
            AuthorizationFailed
            AuthenticationFailed
            Timeout
            NoSuccess
  Notes:    - if the contact node could not be contacted,
              an 'IncorrectState' exception MUST be thrown.
            - the returned client MUST contain the
              application-specific 'message' object that was
              set by the node that handed off the client.

- close
  Purpose:  Closes all streams of all clients handled by the local
            daemon and cleans up the daemons state.
  Format:   close          (in  bool   binding_reset);
  Inputs:   binding_reset:  whether to clear all bindings of all clients
                            handled by the current node or not
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
            PermissionDenied
            AuthorizationFailed
            AuthenticationFailed
            Timeout
  Notes:    - all subsequent method calls on this object MUST
              throw an 'IncorrectState' exception, except for the
              CONSTRUCTOR, DESTRUCTOR and close().
```

```
            - if binding_reset is true, all clients will connect
              to the contact node when they start a new connection.
```

Class `ds_client`

A `ds_client` object contains all streams that are connected with a particular client. It can also contain an application-specific message (a SAGA buffer) that is passed to the target of a handoff operation, or has been received from another node during a handoff operation.

There are two ways to retrieve a `ds_client` object:

1. via `ds_service.get_client()`, using one of its associated streams returned by `ds_service.serve()`
2. via `ds_service.receive_handoff()`

In the second case, the `ds_client` may contain a message that was set by the node that handed off the client.

A `ds_client` object is a shallow copy of a part of the state of the local Distributed Servers daemon. In particular, the object will not contain any new streams that arrived after it was constructed. Hence, a `ds_client` object simply acts as the identifier of a remote node. Each call to `ds_service.get_client()` may return a new object with independant state. The message contained in a `ds_client` is also local to that particular instance.

```
- DESTRUCTOR
  Purpose:  destroys the object
  Format:   DESTRUCTOR  (in  ds_client  obj);
  Inputs:   obj:         the object to destroy
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the client is closed
  Perms:    -
  Throws:   -
  Notes:    - if the client was not closed before, the
              destructor performs a close() on the instance,
              and all notes to close() apply.

- get_url
  Purpose:  returns the url that identifies the client
  Format:   get_url (out  saga::url  obj_url);
  Inputs:   -
  InOuts:   -
  Outputs:  obj_url: url of the client
```

```
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
            IncorrectState
            NoSuccess
  Notes:    - the URL MUST consist of the scheme 'ipv6://'
              followed by the IPv6 address of the client.

- get_streams
  Purpose:  returns all streams connected with this client
  Format:   get_streams(out  array<saga::stream>  streams);
  Inputs:   -
  InOuts:   -
  Outputs:  streams: list of streams connected with this client
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
            IncorrectState
            NoSuccess
            PermissionDenied
  Notes:    - the returned array is a shallow copy; any subsequent
              streams connected with this client that are served
              later will not be included in the array.

- set_message
  Purpose:  sets application specific data that will be sent
            along with a handoff
  Format:   set_message  (in     saga::buffer     msg);
  Inputs:   msg:          buffer containing application-specific
                          data, or NULL.
  InOuts:   -
  Outputs:
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
            IncorrectState
  Notes:    - any message set previously will be overwritten
            - using NULL as a message effectively removes it

- get_message
  Purpose:  returns the application-specific data of this client
  Format:   get_message  (out  saga::buffer  msg);
  Inputs:   -
  InOuts:   -
  Outputs:  msg:          the application-data associated with
                          this client
```

```
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
            IncorrectState
  Notes:    - the data can have been set by another node that
              handed off this client.
            - if no data has been set, NULL MUST be returned.
```

Interface `handoff_policy`

A `handoff_policy` chooses one target Snode from a set of possible targets. An implementation of such a policy must be provided to a `ds_service` object, which will use it for all handoff operations. Handoff policies can be very application-specific.

```
- get_target
  Purpose:  returns the URL of the Snode a client should be
            handed off to.
  Format:   get_target  (in   ds_client   client,
                         in   array<url>  options,
                         out  saga::url   target)

  Inputs:   client:      the client object to select a target for
            options:     the possible  targets to select
  InOuts:   -
  Outputs:  target:      the selected target
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   DoesNotExist
            NoSuccess
  Notes:    - if there are no possible targets, a 'DoesNotExist'
              exception MUST be thrown.
```

A.2 Example

The following Java code example demonstrates the basic usage of the XOSAGA API for Distributed Servers. The methods in the example may sometimes deviate slightly from the specification, which indicates the use of default values (e.g. the 'timeout' parameter in `ds_service.serve()`).

The example consists of three classes: `PickFirstHandoffPolicy`, `HandoffDonator`, and `HandoffReceiver`.

The class `PickFirstHandoffPolicy` implements a very simple handoff policy: always return the first option.

```
class PickFirstHandoffPolicy implements HandoffPolicy {

    public URL getTarget(DsClient client, List<URL> options)
    throws DoesNotExistException {
        if (!options.isEmpty()) {
            return options.get(0);
        } else {
            throw DoesNotExistException("No possible targets");
        }
    }

}
```

The class `HandoffDonator` continuously listens to incoming streams. After each stream is used in some application-specific code, the client of the stream (i.e. its source) is handed off to another Distributed Servers node using the pick-first handoff policy. The handoff is accompanied by a SAGA Buffer object that contains information about the reason for the handoff operation.

```
class HandoffDonator {

    public static void main(String args[]) {
        DsService ds = null;

        try {
            Session def = SessionFactory.createSession(true);
            HandoffPolicy p = new PickFirstHandoffPolicy();
            ds = new DsService(def, "default", p);

            while (true) {
                Stream s = ds.serve();

                // <application-specific actions>

                DsClient client = ds.getClient(s);
                Buffer msg = createHandoffMessage();
                client.setMessage(msg);

                try {
                    URL target = ds.handoff(client);
                    System.out.println("Client " + client
                            + " handed off to " + target);
                } catch (SagaException e) {
                    System.out.println("Handoff failed: "
                            + e.getMessage());
                    s.close();
                }
```

```
            }
        } catch (Exception e){
            System.out.println(e.getMessage());
        } finally {
            if (ds != null) ds.close();
        }
    }

}
```

The class `HandoffReceiver` waits until some other Distributed Servers node performs a handoff operation. It will then extract the handoff message and perform some application-specific actions. Since it will never hand off clients itself, no handoff policy is provided to the `DsService` object.

```
class HandoffReceiver {

    public static void main(String args[]) {
        DsService ds = null;

        try {
            Session def = SessionFactory.createSession(true);
            ds = new DsService(def, "default", null);

            DsClient client = ds.receiveHandoff();
            Buffer msg = client.getMessage();

            // <application-specific actions>

        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            if (ds != null) ds.close();
        }
    }
}
```
