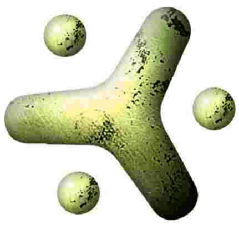

# Jtrix platform specification

12th September 2001

Ulf Leonhardt
mailto:feedback@jtrix.org
Jtrix Ltd
57-59 Neal Street, London WC2H 9PJ, UK
+44 20 7395 4990

Objectives of this document:

- To define interfaces and behaviour that Jtrix node implementations must support.
- To define interfaces and behaviour that Jtrix Netlets must supports in order to be accepted and run by Jtrix nodes.
- To define exchange formats for transferring services and netlets between nodes
- To enable multi-vendor interoperability between netlets and nodes

To this end, this document defines the structure and interactions of entities that were identified/described in the Jtrix Technical White Paper[1]. A separate document will describe the reference implementation of a node.

We welcome feedback and suggestions through the Jtrix web site `http://www.jtrix.org/`. There you also find documentation of the Jtrix API.

[1] Available from the Jtrix web site `http://www.jtrix.org/`.

# Contents

# List of Figures

# List of Algorithms

## Glossary

**Access point** The netlet or node providing local access to a service.

**Accounting group** Set of netlets whose resource charges are handled together.

**Binding URL** A URL for invoking a binding protocol.

**Binding** See service binding.

**Binding protocol** A protocol for obtaining a netlet descriptor for a service access point.

**Certificate** In this paper, a signed document binding a Public Key to a name.

**Codebase** A collection of Jar files containing all the netlet's classes.

**CRL** Certificate revocation list, a list of signed revocation statements.

**Descriptor** See netlet descriptor.

**Distinguished name** An X.500 name.

**DTD** XML document type definition.

**Facet** An alternative interface of an object that is obtained by querying the primary interface.

**Hosting service** A service provided by a collection of nodes to allow remote netlet execution.

**Hosting contract** The agreement under which a netlet executes on a node and is charged for it.

**Jar** Java archive file format[2].

**Jtrix** The Java matrix—an open, Internet-wide collection of nodes and netlets.

**JVM** The Java virtual machine

**Mediator** A node-local communication object that exports a single interface across a netlet boundary.

**Netlet** An executable, context-independent component in Jtrix.

**Netlet descriptor** An XML document given to a node to create a netlet.

**Node** A local environment where netlets can execute and exchange services.

**Parameter bean** A container for netlet creation parameters.

**Principal** The holder of a public/private key pair.

**Resource** A node-dependent commodity, often hardware-related (e.g. CPU, Memory, etc.)

**Service** A node-independent commodity. A shared object that can be accessed from any Jtrix node.

**Service binding** The process of establishing a service connection.

**SHA** Secure Hashing Algorithm (NIST standard, also known as SHA-1).

**Warrant** An XML documents that represents a right to use a particular service.

**X.500** A CCITT[3] standard for distributed directories.

**X.509** A CCITT standard for digital certificates.

**XML** Extensible markup language, as recommended by W3C[4].

[2] `http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html`
[3] `http://www.itu.int/`. Unfortunately, they charge for downloading of standards documents.
[4] `http://www.w3.org/TR/2000/REC-xml-20001006`

# 1 Introduction

## 1.1 What is the Jtrix platform?

The Jtrix platform is a framework for implementing an open, distributed execution environment for software components. It defines three basic entity categories in the system:

**Netlets** (Section 9) are code-mobile application components.

**Nodes** (Section 6) are executors of netlets, typically with a fixed physical location.

**Services** (Section 4) are location-independent providers of functionality and/or state.

Netlets, nodes, and services are required to support certain protocols and interactions, as well as certain document formats (e.g. descriptors and warrants, see Section 11).

A node provides a local component execution environment that offers certain facilities and guarantees to netlets executing within the environment, particularly with respect to:

- component integrity (Section 8)
- service binding (Section 4.8)
- versioning (Section 9)
- resource accounting and control (Section 10)

## 1.2 Design principles

In designing the functionality of the platform we were guided by the following principles:

- End-to-end argument[5]
  - Small execution platform - the system is extended by adding services.
  - Very small set of required network protocols, other protocols can be used freely as long as the endpoints agree.
  - No prescribed contracts
  - No prescribed electronic payment mechanism or currency.
- No central authorities
- Openness
- Extendability

## 1.3 Terminology

This specification follows the guidelines set out in RFC 2119[6] for defining the significance of each particular requirement:

**MUST** This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

[5] `http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.txt`
[6] `http://sunsite.org.uk/computing/internet/rfc/rfc2119.txt`

**MUST NOT** This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

**SHOULD** This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

**SHOULD NOT** This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behaviour is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behaviour described with this label.

**MAY** This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

An implementation is not compliant with this specification if it fails to satisfy one or more of the MUST requirements for the protocols/interfaces it implements.

# 2 Netlet

## 2.1 Function

In Jtrix, netlets collectively contain the logic and state of services and applications.

More specifically, a netlet can perform the following actions:

- Execute its own code in a node-provided sandbox.
- Request and use resources and facilities from the node.
- Bind and use services mediated by the node.

The sandbox enforces the separation of netlets and prevents uncontrolled usage of node resources. For example, the sandbox would prohibit or redirect I/O related functions of the Java API.

It is mandatory that netlets correctly implement the `INetlet` interface.

The netlet's descriptor (Section 2.10) and codebase (Section 2.7) must be made available to the node. This may involve making the Jar files available for downloading from the network.

## 2.2 Identification

There is no general netlet identification or netlet addressing scheme in Jtrix.

However, nodes may assign identifiers to netlets for administrative purposes.

## 2.3 Run-time structure

A netlet executes inside a single JVM. It consists of the following parts:

- a set of loadable classes, i.e. the netlet's codebase.
- a set of instantiated objects inside the JVM.
- a set of running threads inside the JVM.

### 2.3.1 Object management

The node references (and therefore keeps alive) the netlet's root object (implementor of `INetlet`).

### 2.3.2 Thread management

To serve an outside invocation of a netlet method, the node must allocate a thread in which the method is executed. Netlets should avoid blocking these threads as the number of threads per netlet may be limited.

Additionally, netlets may create their own threads.

## 2.4 Interface to the node

Interaction between netlet and node is carried out through a session between `INetlet` and `INode` interfaces (`org.jtrix.base`). Both sides are also facet providers to allow for additional interfaces to be negotiated.

A netlet must implement the `INetlet` interface (cf. Figure 1). This allows the node to request state changes by the netlet, and to inform the netlet of certain events.

`INetlet.bindService()` is used by the node to bootstrap service connections. It is only used if the netlet was instantiated as a service access point (Section 4.5) during a bind operation.

The netlet interface is bound to the node when the netlet is instantiated (see 2.8.2).

The INode interface is described in Section 3.3.

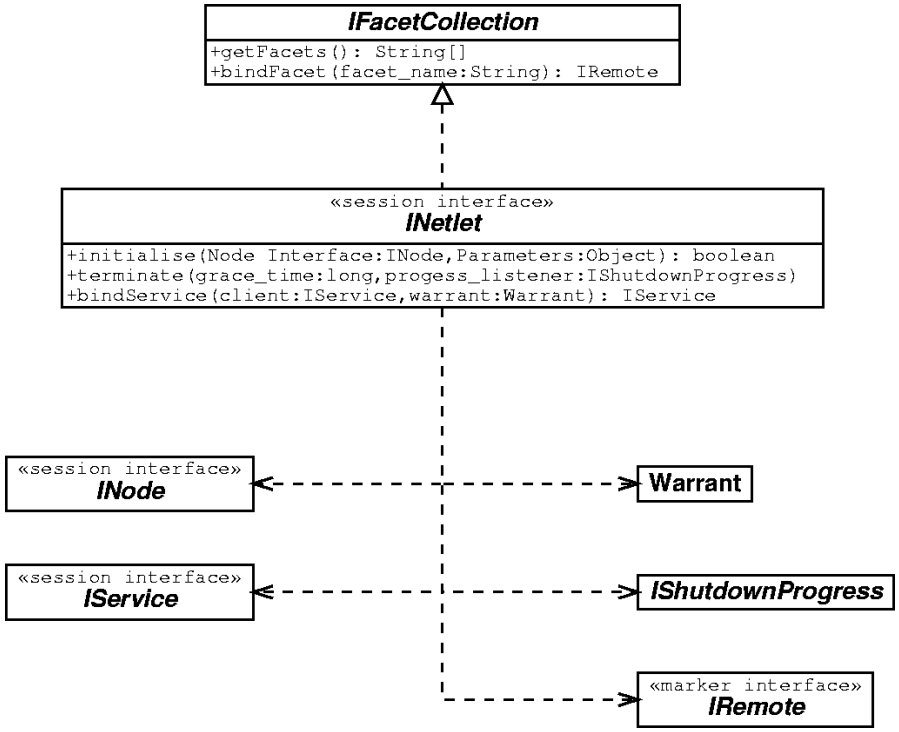


*Figure 1: Netlet interface*

## 2.5 Netlet facets

There are no mandatory facets for netlets.

Optional netlet facets:

**`org.jtrix.facet.netlet.ITextStatusFacet`** Allows the node to retrieve simple status information from the netlet.

**`org.jtrix.facet.netlet.ITypedStatusFacet`** Allows the node to retrieve a typed tree with status information from the netlet.

## 2.6 Accounting group

An accounting group is a group of netlets sharing the same hosting contract and payment mechanism. Each netlet is in exactly one accounting group. A netlet's accounting group is fixed for the lifetime of the netlet.

### 2.6.1 Lifetime

The lifetime of an accounting group determined by the associated hosting contract. When a group is deleted from a node, all the netlets in it are terminated by the node.

### 2.6.2 Extension

A netlet can be created in an existing accounting group. This happens when a local service access point is instantiated (see Section 4.8). Other mechanisms for inserting a netlet into an accounting group may exist but are outside the scope of this specification.

### 2.6.3 Reduction

When a netlet terminates, it automatically leaves its accounting group.

## 2.7 Codebase and class loading

The netlet's descriptor defines the Jar files available to the netlet. Therefore, it determines which classes a netlet can use.

When a netlet is initialised all its Jars labelled as non-lazy in the descriptor must have been downloaded already. Remote Jar files are retrieved via HTTP or HTTPS.

If a netlet refers to a class outside its codebase, a `java.lang.ClassNotFoundException` is thrown.

## 2.8 Lifecycle

### 2.8.1 States

Figure 2 shows the states of a netlet from the perspective of its host node. The states are characterised as follows:

**Loaded** The netlet's non-lazy Jar files are in possession of the node. The node's internal objects for managing the netlet have been created.

**Running** The netlet's object are inside the JVM's object space. Threads can be active inside the netlet.

**Terminated** All netlet threads are terminated, and no new threads can be created. Node objects for the netlet may still exist.

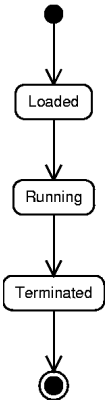


*Figure 2: Netlet states*

### 2.8.2 Instantiation

Algorithm 1 shows the steps the node follows when creating a new netlet (see also Figure 3).

1. Load non-lazy Jars from the codebase. If a Jar cannot be retrieved, abort the netlet instantiation.
2. De-serialise the signed parameter bean from the descriptor (if present). If parameter bean cannot be instantiated from the netlet's codebase, abort the netlet instantiation.
3. Set the `argument` property of the parameter bean to the unsigned argument (if present). If the property cannot be set, abort the netlet instantiation.
4. Create a new instance of the netlet's main class. If the netlet instance cannot be created, abort the netlet instantiation.
5. Call `INetlet.initialise()`. If `initialise()` returns `true`, mark the netlet as running. Otherwise, initiate the netlet termination sequence.

*Algorithm 1: Netlet instantiation*

The Jtrix platform does not specify a direct interface for triggering netlet instantiation, although service binding involves creation of netlets. However, standardised interfaces for remote creation of netlets should be provided by higher-level services in association with the node.

### 2.8.3 Termination

A netlet may terminate voluntarily, or it may be asked by the node to do so (see Figure 4). In either case, it is recommended that netlets terminate all their threads. After all the threads have terminated, the node should remove the netlet and release all corresponding resources (memory, classloaders, mediators, etc).

The node can request a Netlet's termination at any time *after* the Netlet has successfully initialised (ie. after `initialise()` has returned `true`). If the Node wants to terminate the Netlet during the initialisation phase, it must not call `terminate()`. Instead, it should kill all threads belonging to the Netlet immediately.

Should a netlet be unable/unwilling to terminate all its threads in a timely fashion, the hosting node should terminate the remaining threads.

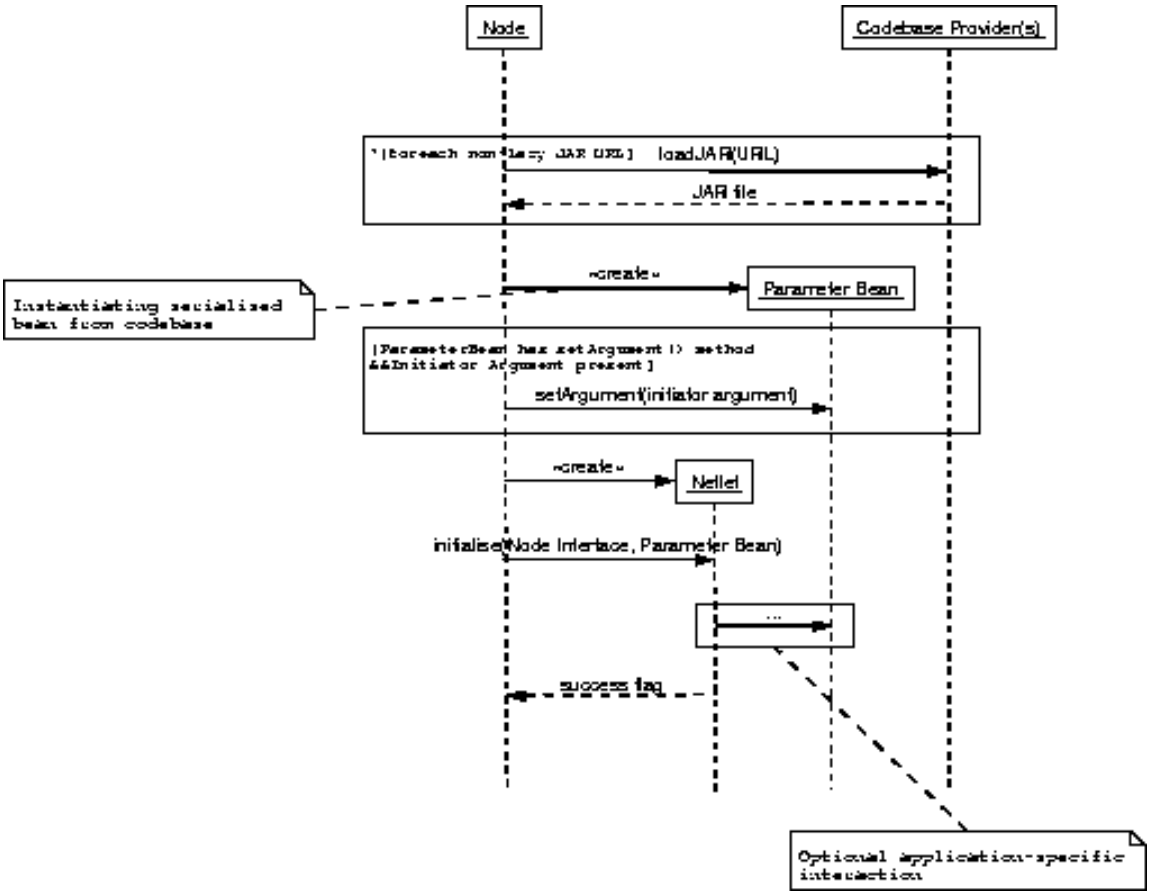


*Figure 3: Netlet instantiation sequence*

There is an informational facility for netlets to publish their shutdown progress as a percentage completed. This does not affect the shutdown sequence.

Figure 4(A) shows the scenario where termination is triggered by the node. This could happen, for example, when the node is shut down.

Figure 4(B) shows a netlet asking to be shut down. After the initial `requestTermination()` request, this sequence is identical to case (A).

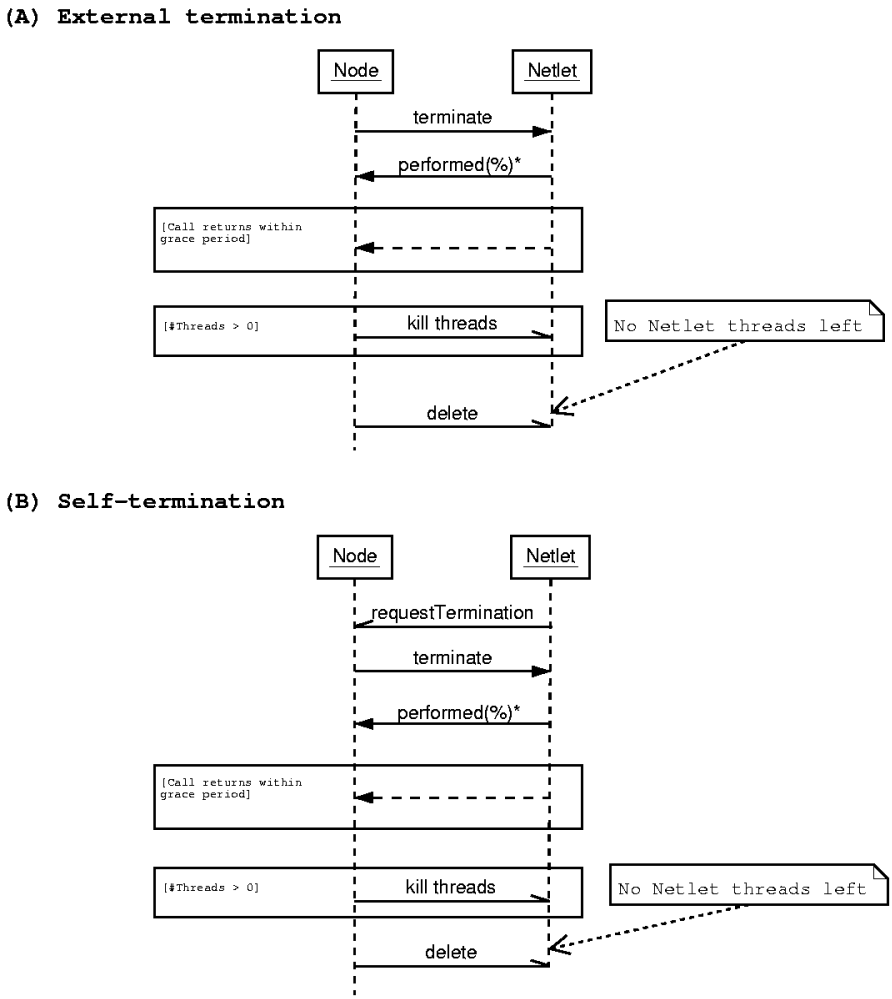


*Figure 4: Netlet termination sequence*

## 2.9 Secrets

A secret is a capability that gives the holder of the secret the permission to take control of a running netlet, e.g. kill it, debug it, monitor its status. A netlet's secrets are listed in its descriptor.

Depending on the algorithm specified in the descriptor, a secret can be either a public key or an unencrypted string. The node must verify (i.e. challenge) any claims by netlets that they are party to a secret before granting them access.

## 2.10 Netlet descriptor

A netlet descriptor (often just referred to as descriptor) is a capability to instantiate a netlet.

The structure of a netlet descriptor is shown in Figure 5 (XML DTD in Section 11.1). The salient components are:

**Codebase(s)** consists of the name of the main class, a set of Jar entries, and a serialised parameter bean. Each Jar entry has a bunch of URLs locating the actual Jar file, as well as the Jar's hash value.

**Signature(s)** allow verification of the descriptor content (excluding the actual Jar file). The signature consists of the signer's distinguished name, and an encrypted digest of the descriptor content. Note that descriptors used to instantiate access points for non-anonymous service must be signed by the service's principal.

**Secret(s)** as described above (Section 2.9).

**Code file(s)** are actual Jar files embedded in the descriptor. Each file is marked by a URL, which ties the Jar to a codebase entry. Embedded Jars are useful to avoid additional network transactions for downloading a netlet's Jar files.

Node implementations must not instantiate netlet descriptors whose lifetime has expired.

Node implementations must treat netlet descriptors as confidential information (as they may contain secrets).

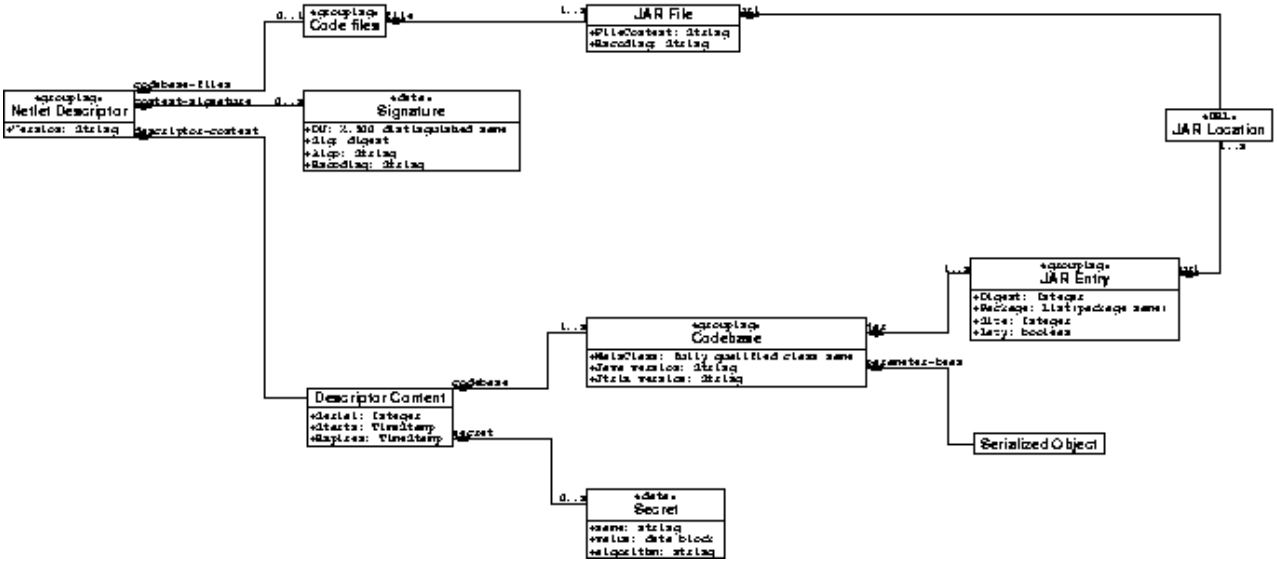


*Figure 5: Netlet descriptor structure*

# 3 Node

Logically, a node is a location for netlets in the Jtrix universe. Only netlets on the same node can converse directly using Jtrix mechanisms.

A node is also an administrative domain, i.e. it defines a set of administrative policies.

Typically, a node would span one or more Java JVMs on one or more computer. It is also possible to run multiple nodes inside the same JVM, although this seems useful only for simulation and demonstration purposes.

## 3.1 Functions

A Jtrix node has the following mandatory functionality:

- create and execute netlets from netlet descriptor
- download netlets' Jar files if necessary
- process bind requests and warrants
- implement the standard binding protocols
- mediate connections between netlets and between itself and netlets
- dispatch invocations of the netlets' interfaces
- enable the auditing of service connections between netlets
- provide resources to netlets as specified by their hosting contract
- protect the availability, integrity and confidentiality of netlets and netlet descriptors to the extend guaranteed by the hosting contract
- enforce access control to netlets based on netlet secrets
- verify certificates, signatures (on warrants and descriptors), secrets with due diligence

In addition, nodes should provide the following facilities:

- allow foreign netlets to migrate onto the node from other nodes through a standardised hosting service
- meter the use of system resources by netlets and charge for their use as specified by the hosting contract
- provide local interfaces (facets) to netlets for debugging, auditing, control

## 3.2 Identification and authentication

Jtrix does not mandate a node identification scheme. However, nodes should be identifiable for diagnostic purposes.

A node may be under the control of a principal, in which case the principal's credentials should be used by the node.

## 3.3 Interfaces

Nodes must support the node interface (`org.jtrix.base.INode`) which is visible to all netlets (Figure 6). The node may provide additional *facets* for administration and debugging (which should be standardised at some point).

A node can vary its facets for different netlets.

### 3.3.1 Facets

**`org.jtrix.facet.node.ITrampolineFacet`** Allows the Netlet to control the providers for file and network I/O used by the local Java API.

**`org.jtrix.facet.node.IClusterFacet`** Allows Netlets to communicate in groups and elect group leaders.

**`org.jtrix.facet.node.INodeResourceFacet`** Provides access to locally available resources.

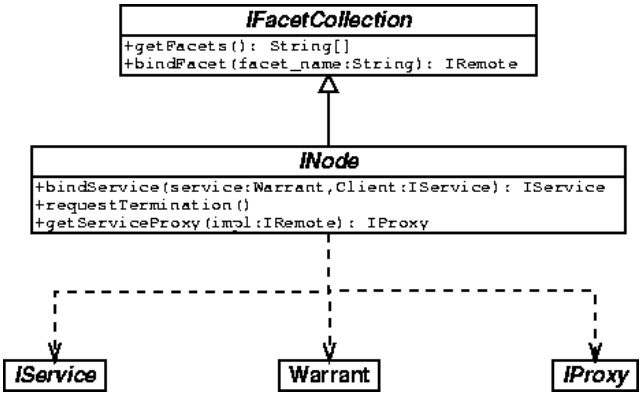


*Figure 6: Node interface*

## 3.4 Lifecycle

Figure 7 shows the life-cycle of a node. The states can be described as follows:

**Inert state** No netlets are present. For example, the machine might be switched off.

**Running state** Normal operational state when netlets are running, and new netlets can be created.

**Shutting-down state** Transitional phase where netlets have been given time to shut down, but no new netlets can be created.

Netlets are not preserved through inert periods.

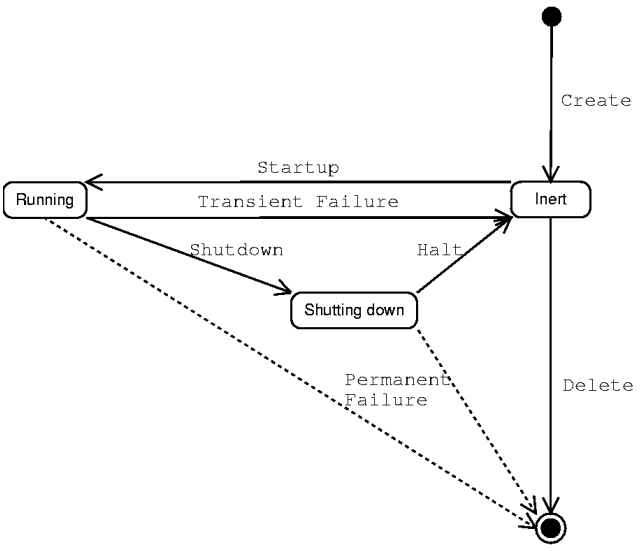


*Figure 7: Node life-cycle*

### 3.5 Support for services

The node must fulfill bind requests as described in Section 4.8.

To this end, a node must keep track of all service access point netlets that it hosts. For each it must record the warrant used to initially bind the access point.

### 3.6 Helper netlets

Node implementations may delegate functionality to helper netlets.

The node is free to instantiate as many such netlets as it sees fit. The node can re-export facets of helper netlets as part of the node interface to other netlets.

### 3.7 Hosting contract

A hosting contract is the contract that obliges the node to run a netlet. The platform does not specify its shape or form, but implies its existence.

### 3.8 Netlet isolation

The node must ensure that netlets can fail independently, are accounted for, and do not corrupt others. Therefore, the node must implement complete isolation of netlets running on it. This entails:

- *Netlets cannot share threads.* This ensures that all threads belonging to a netlet can be removed without damaging others.
- *Netlets cannot share netlet-writable memory.* This ensures that allocated memory can always be attributed (and charged) to a single netlet.
- *Netlets cannot share netlet-writable namespaces.* This ensures that netlets cannot masquerade each others' classes.
- *Node threads, memory, namespaces cannot be written to by netlets.* This protects the node from malicious netlets.

See Section 8 for detail on protection mechanisms.

## 4 Service

A service is a universally available entity that netlets (and nodes) can communicate with.

### 4.1 Functions

A service must support the following functionality:

- Issue warrants (directly or via agents) for use of the service.
- If issued warrants contain binding URLs, arrange for availability of corresponding binding protocol listeners, and Jar files.

## 4.2 Identification and authentication

A service can be a principal. If this is the case, it is identified by the principal's X.500 distinguished name (DN) and public key.

A service without a principal is known as an anonymous service. There is no built-in authentication. However, a service can be identified by the Warrants it has issued. Any warrant has only one service which it can bind to.

## 4.3 Interfaces

A service session is composed of a symmetrical pair of IServiceIService interfaces (see Figure 8). Figure 9 shows a scenario with a service provider and and service client.

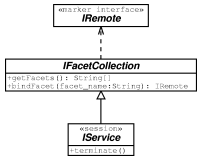


*Figure 8: Service interface*

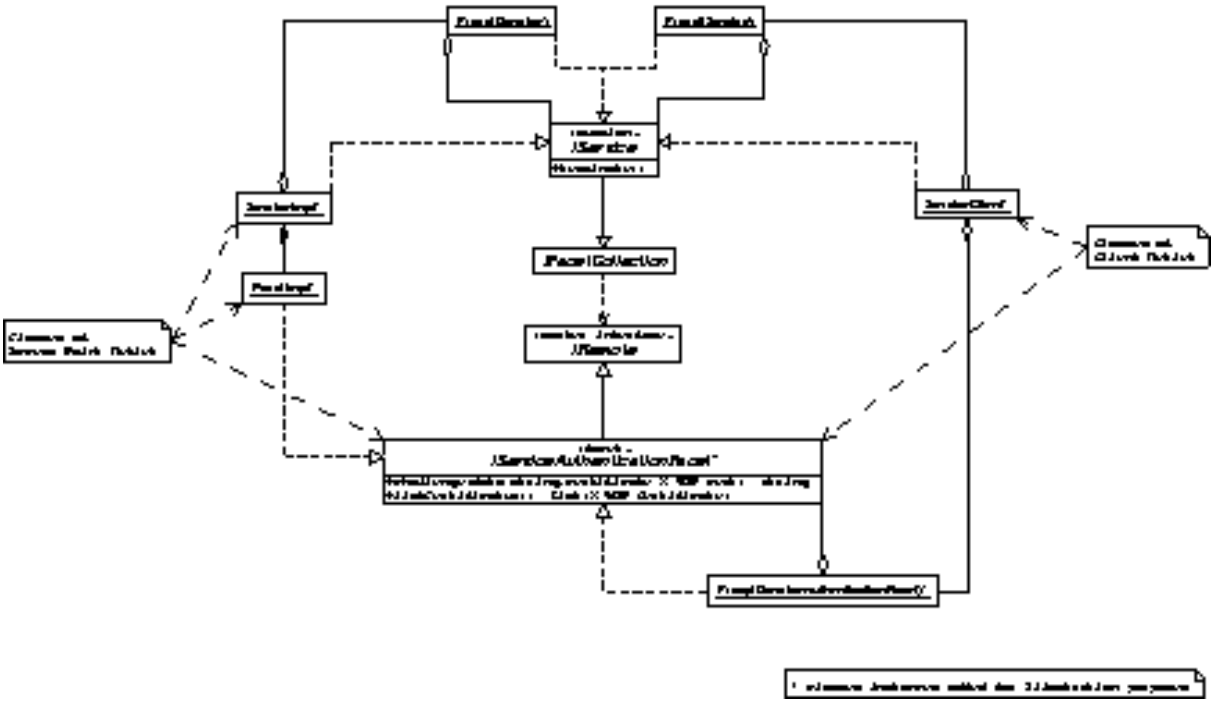

*Figure 9: A netlet-to-netlet service scenario*

### 4.3.1 Facets

All facets on a service connection are service-specific. While standardised services will entail the definition of mandatory and optional facets for each such service, this is not part of the Jtrix platform specification.

## 4.4 Client role

In order for a prospective client component (netlet or node) to use a service, it must communicate with a local access point (netlet or node).

The client has the responsibility to initiate the binding process by presenting a warrant to the node.

## 4.5 Access point role

A service is accessed through a component (netlet or node) on the same node as the client component.

This component is said to be an access point for that service. A component can be an access points for more than one service, and many components can be access points for the same service.

An access point's responsibilities are twofold:

1. Process bind requests for the service, establishing new service sessions if necessary.
2. Serve existing service sessions.

The node decides whether and how bind requests are routed to existing access points. The access point can be internal or external. The choice is made during the initial binding of the access point.

### 4.5.1 Internal access point

An internal access point netlet executes in an accounting group with hosting contract established by the client. If the client is a netlet, this would be the client netlet's accounting group.

This implies that the lifetime of a local access point is limited by the client's hosting contract.

### 4.5.2 External access point

An external access point (node or netlet) executes in its own accounting group which must be established when the access point is first created. External access point netlets can only be created by a binding protocol that can establish a new hosting contract.

An external access point can receive bind requests from any netlet on the node, thus allowing for more efficient use of resources. Because it has its own hosting contract, it can request a quality of service level from the node according to its needs. Also, the access point's lifetime is independent of any of its clients.

## 4.6 Location

Jtrix services are conceptually location-independent. Hence, Jtrix is unaware of the location of a service as such. Implementations of access points to a service (i.e. service netlet descriptor) are found by the node through binding URLs contained in a warrant.

Binding URLs can use established protocols, such as HTTP, or new protocols specifically designed for Jtrix.

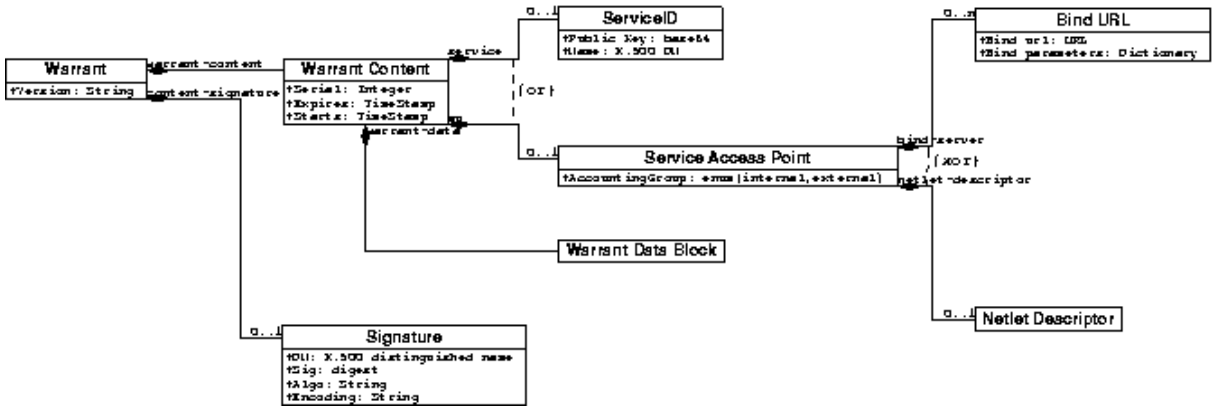


*Figure 10: Warrant structure*

## 4.7 Warrant

A warrant is a bearer-instrument to establish a service binding. Figure 10 shows the composition of a warrant. Jtrix defines an XML representation for warrants (see Section 11.1).

A warrant has the following salient components (cf. Figure 10):

**Access point** (optional) is either an embedded descriptor or a set of binding URLs which can be used to create a new access point. Optionally, it specifies the account type (internal or external).

**Service ID** (optional) is the principal's name and public key. Its presence permits the node-wide reuse of the access point as well as the verification of the netlet descriptor.

**Warrant data** (optional) is application-specific data presented to the access point as part of the bind request.

**Signature** (optional) can be used to verify the authenticity of the warrant.

A Warrant that does not have access points or service IDs cannot be used to bind a service.

Inside a node, warrants can be passed around without being converted into to XML (using `org.jtrix.base.War`

A warrant is invalid if a Signature is present unless `service-id/public-key` is also present and the Signature can be verified with `public-key`.

## 4.8 Binding sequence

Service binding is the process of establishing a session with a service from a warrant.

As shown in Figure 11 and Figure 12, the client's node plays a central role during service binding. Binding proceeds as shown in algorithm 2.

No assumption should be made by netlets that two bind request to the same service arrive at the same access point

## 4.9 Binding protocols

A binding protocol is employed by the node to obtain a netlet descriptor for the creation of a service access point. The protocol is invoked with a binding URL found in the warrant (see Figure 10).

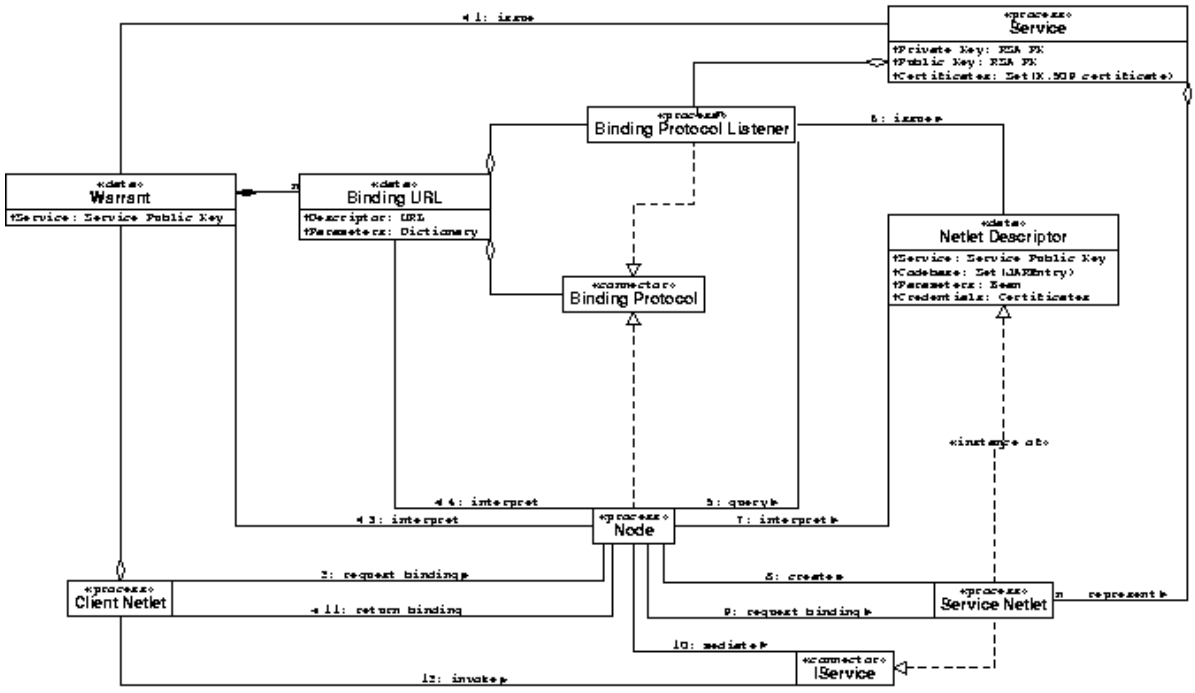


*Figure 11: Binding overview*

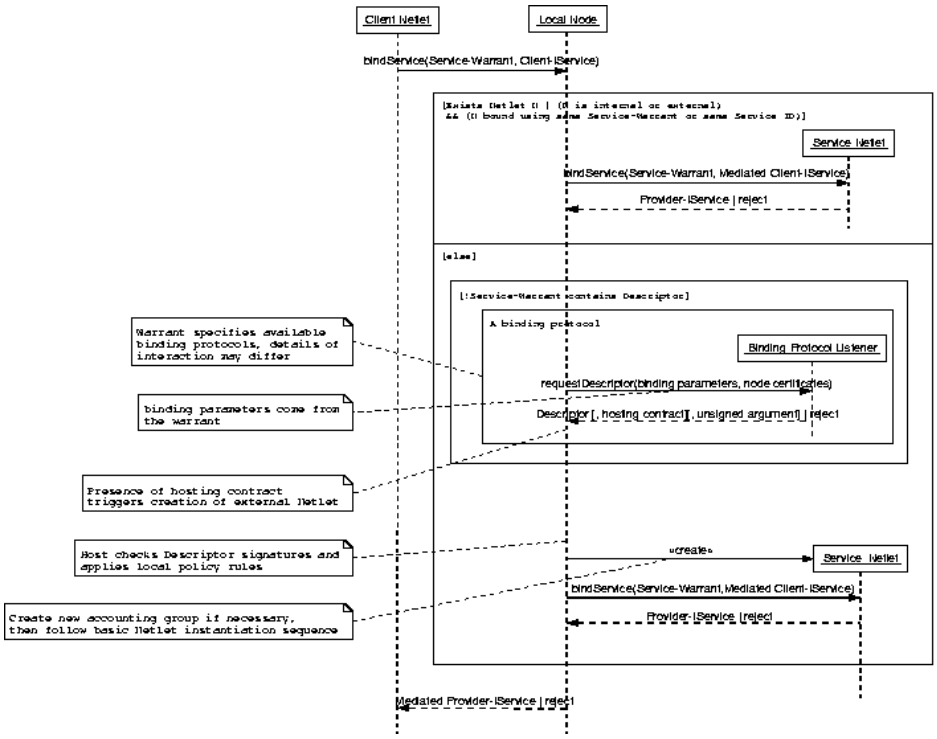


*Figure 12: Binding interaction*

1. Service issues warrant which is propagated to the prospective client.
2. Client asks node to bind warrant.
3. Node interprets and verifies warrant. If usable access point exists (see Algorithm 3), shortcut to step 9. If warrant contains descriptor, shortcut to step 8.
4. Break out binding URL from warrant and look for locally available binding protocols.
5. Invoke binding protocol, present credentials to binding server if required by binding protocol.
6. Binding server issues netlet descriptor for service access point to node.
7. Node interprets and verifies netlet descriptor. In order to satisfy the binding of a signed warrant with a service ID, the descriptor must be signed by the same principal. Otherwise, the service binding fails.
8. Node instantiates access point netlet from descriptor.
9. Node requests binding from access point netlet. If pre-existing access point declines request, mark access point as unusable and go back to 3.
10. Node mediates service session interface returned by access point netlet.
11. Node returns mediated service session interface to client.

*Algorithm 2: Binding algorithm*

An access point netlet that was initially created with warrant $W_1$ may be reused to bind warrant $W_2$ if all the following hold:

1. $W_2$ is valid
2. Accounting criteria are valid
3. Service criteria are correct

Otherwise the existing access point netlet is ignored.
Service criteria are correct if any of the following hold:

1. Service ID is present and the same in both warrants
2. Warrants are identical

Accounting criteria are valid if any of the following hold:

1. Access point netlet is External
2. Access point netlet is Internal and in same group as client netlet

*Algorithm 3: Decide whether access point is reusable*

Each node must implement the minimal binding protocol specified below (4.9.1).

Nodes are free to offer other, more sophisticated binding protocols, some of which may become standardised. Such a protocol could include some of the following features:

- transfer of hosting contract for new netlet.
- transfer of unsigned arguments for new netlet.
- verification of node's credentials before deploying an access point.
- authentication of binding server before sending the bind request.
- secure network transport.
- negotiation of alternative descriptors depending on the target node and the request parameters.

### 4.9.1 A minimal binding protocol

The protocol simply retrieves a netlet descriptor via HTTP or HTTPS from a binding server. It can only create internal access points.

**URL** A normal URL as specified in RFC1738, with HTTP or HTTPS as protocols.

**Request** The request is a GET or a POST request (see RFC 2068). Binding parameters are sent in the proper encoding as part of the URI or the request body, respectively.

**Response** A response is valid if, and only if, all of the following holds:

1. The response is a valid HTTP response.
2. The response code is "200 OK".
3. The response body is an XML-encoded netlet descriptor.

# 5 Naming and certification

## 5.1 Principal

A Jtrix principal is the owner of a public/private key pair. The role of a principal in Jtrix loosely corresponds to the idea of a legal person.

Names are bound to principals. A principal must not have more than one name.

A Jtrix service can be a principal. It is unspecified, however, how nodes relate to principals.

## 5.2 Names

Names in Jtrix are X.500 distinguished names. The Jtrix platform can function without names, but names should make a large-scale deployment more managable.

## 5.3 Certificates

In Jtrix, a certificate is a binding between a principal's public key and its X.500 distinguished name. The certificate must be signed by a principal, who is referred to as the certifying principal. A principal may certify itself.

The certifying principal should take reasonable steps to ensure:

1. X.500 name properly corresponds to certified principal.
2. X.500 name is probably unique and will probably remain so.

### 5.3.1 Certificate revocation

RFC 2459 proposes "CRL Distribution points" as a "non-critical" extension to the X.509 certificate format. A certificate using this extension contains a list of CRL URLs.

Certifying principals should use the CRL extension and make CRLs available.

Node implementations may check CRLs when a certificate is presented. Node implementations may also reject certificates without declared CRL distribution points as a matter of policy.

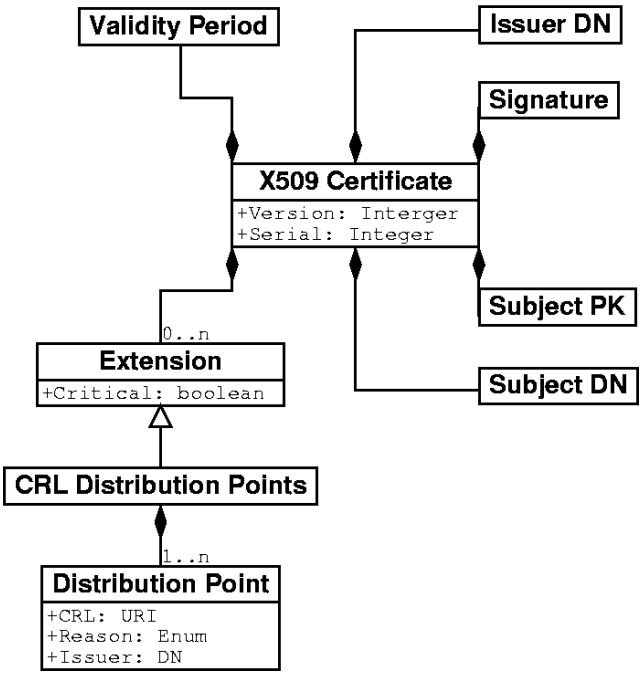


*Figure 13: X.509 certificate structure with CRL distribution point extension*

### 5.3.2 Certificate transport

Jtrix does not contain a certificate transport at the platform level. Higher-level entities, such as binding protocols and hosting services, are free to specify their own certificate transport mechanism.

## 5.4 Identification

A principal is uniquely identified by the pair (Public Key, X.500 DN). A collision of this pair is equivalent to a compromise of the private key and must be treated as such (i.e. renew key pair). Collisions of the X.500 DN can be tolerated, although they should be minimised for efficiency's sake.

It is legitimate for entities in Jtrix to refuse identities that are either uncertified or certified by "untrusted" principals.

## 5.5 Authentication

A principal is authenticated by challenging its private key using the public key.

## 5.6 Signatures

A signature associates an artifact (descriptor, warrant, certificate, Jar) with a principal. It indicates that the principal either constructed the artifact, or approved of its construction.

| Role of principal | Signed artifacts |
|---|---|
| Service operator | Warrant, descriptor |
| Netlet developer | Descriptor, Jar files |
| Netlet owner | Descriptor |
| Certification agency | Descriptor, Jar files, certificates |

Signature verification requires presence of the signer's public key. Keys are distributed in warrants, and optionally, in X.509 certificates (see Figure 13). Keys are transmitted, for example, during the binding process.

The certificate used for verification is tied to the signature via the principals X.500 DN.

# 6 Communication sessions

There are some patterns of interaction common to the communication between netlets and between netlets and nodes.

## 6.1 Remote interface

A remote interface is an interface to a Java object in a different address space. It is a directional connection between a netlet and another netlet, or between a netlet and a node.

While a remote interface largely looks like a local interface, there are some important differences:

- The interface is marked by `org.jtrix.base.IRemote`.
- The interface may be explicitly closed down.
- User and provider of the interface may fail independently and at any time.
- User and provider have different classloaders and different sets of classes.
- The interface may be accessed and/or implemented asynchronously.
- Arguments, returns, and exceptions behave differently (see 8.4.2).

The Jtrix API provides classes for detecting and exploiting the above differences if the application chooses to do so. For example, `IAsynchronousClient` and `IAsynchronousServer` (both in `org.jtrix.base`) allow thread-economic implementation of users and providers.

## 6.2 Session

A session is a primary connection between two parties, e.g. a netlet and a node. The session is controlled through a session interface at either end. The session may also contain any number of secondary connections via normal remote interfaces in both direction.

Session management is carried out through the session interfaces, which are remote interfaces. They allow for the session to be terminated, and for secondary connections to be created. Session management is symmetrical, i.e. any action can be initiated from both sides. Termination of a session results in disconnection of the session interfaces and all secondary interfaces.

A session interface in Jtrix is concerned with session control only, with actual functionality delegated to *facets*. Therefore, session interfaces are typically *facet providers*.

## 6.3 Facets

A facet is a remote interface obtained from a facet provider. Facets obtained from the same provider are intended to be alternative interfaces to the same underlying state. There is no constraint that a facet provider always return the same facet object.

## 6.4 Facet provider

A facet provider allows access to alternative interfaces for a given implementation object. A facet provider may also enumerate the supported facet types (see Figure 14).

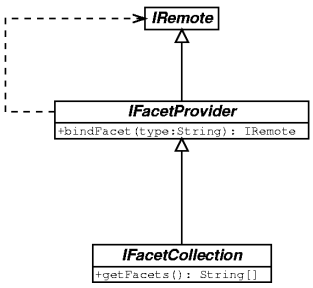


*Figure 14: Facet providers*

## 6.5 Node sessions

Netlet and node interfaces are bound when a netlet is instantiated (Figure 15). Figure 9 shows the interfaces that exist between a netlet and its hosting node.

## 6.6 Service sessions

Services are provided directly by the hosting node (Figure 16), or indirectly by another netlet through the hosting node (Figure 17).[7]

Service Interfaces are bound by the node after presentation of a warrant by the Initiator of the bind operation. The target is always a service, i.e. it is not possible to bind to a specific netlet.

[7] The client netlet cannot tell the difference.

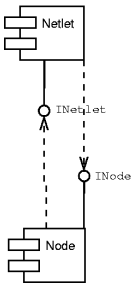


*Figure 15: Basic netlet-node interfaces*

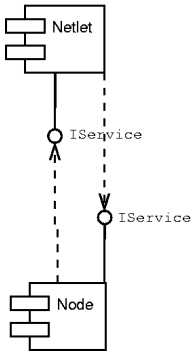


*Figure 16: Node providing a service connection to netlet*

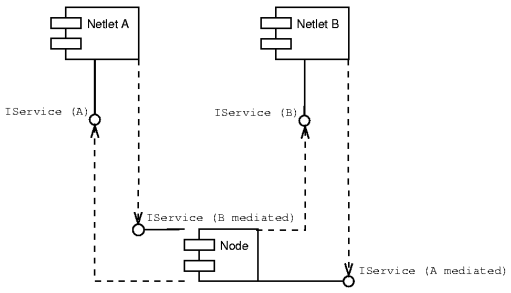


*Figure 17: Mediated netlet-netlet service connection*

# 7 Trust

## 7.1 Assumptions

- *Netlets completely trust their local node.* Netlets are defenceless against their node. Assuming the netlet cares, it must trust the node it is running on.
- *Netlets trust the services they use to some extent.* Otherwise, services would not be useful.
- *Nodes are mutually suspicious.* Nodes can be altered or subverted, hence there is no reason why one node should trust another.
- *Netlets are suspicious of other nodes.* As above.
- *Netlets are mutually suspicious.* Netlets could be written with malicious intent, hence there is no reason why one netlet should trust another.
- *Nodes are suspicious of all netlets.* As above.

## 7.2 Model

Trust in Jtrix is based on the idea of assigning local trust levels to known principals against a background of mutual suspicion.

To be useful, principals must be closely linked to netlets, nodes, and services.

**Netlet** The platform only needs to trust a netlet if it provides node-wide access to a service. In this case, the service is the netlet's principal and has to sign its descriptor.

**Node** Trust in the node can be necessary during service binding. The binding protocol should allow the binding server to authenticate the node before the netlet descriptor is returned. They same is likely for future hosting services. Therefore, it is recommended that nodes are linked to a principal.

**Service** Services can be principals. This is expressed by warrants and descriptors that are signed by the service.

## 7.3 Trust assignment mechanisms

Given a principal-based model, it is critical how one arrives at the trust level for a principal.

### 7.3.1 Blissful ignorance

Trust anyone. This is very effective in a perfect world (read "closed system"). Apart from that, it can make a useful seed for feedback mechanisms.

### 7.3.2 Known principals

Trust a set of known principals. This mechanism cannot assign any trust to unknown principals, and it therefore only useful in conjunction with inductive mechanisms.

#### 7.3.3 Certification of principals (Induction)

If an untrusted principal A present a certificate signed by a trusted principal B, a B must have trusted A to some degree, otherwise B would not have signed the certificate. As a result, a small amount of trust can now be assigned directly to A.

This principle can be extended to signature delegation chains, although it becomes very tentative after only a few levels of delegation.

#### 7.3.4 Feedback

The above mechanisms are all static, i.e. they cannot cope with mistakes and temporal changes in the trustworthiness of principals. To address this, one has to use feedback to adjust the trust levels in the system. It is possible that a complex feedback-based trust system for Jtrix will emerge.

**Positive feedback** Trust should become stronger after a successful interaction with the principal. It could also become stronger if someone else who we trust had a successful interaction with the principal.

**Negative feedback** Is the converse of the positive feedback described above.

# 8 Intra-node protection mechanisms

## 8.1 Memory model

### 8.1.1 Memory sharing

Netlets do not share netlet-created Java Objects. At the node's discretion, netlets may share node-created, immutable Objects (e.g. stateless system classes).

Immutable classes are tagged with `org.jtrix.base.IImmutable`.

### 8.1.2 Memory allocation and lifetime

Netlets are free to create Java objects (subject to node-imposed resource constraints). Memory is garbage collected asynchronously by the Java garbage collector. Memory is freed at some point after the netlet terminates.

## 8.2 Threading model

Threads are private to netlets, e.g. calls between netlets and between node and netlet require a context switch. Netlets can create new threads, subject to node-imposed resource limits. All threads belonging to a netlet are removed when the netlet terminates.

## 8.3 Java namespace

### 8.3.1 Class names

There is a global space for Java class names which is shared between all netlets and nodes. This includes the Java API (e.g. `java.* javax.*`) and the Jtrix API (i.e. `org.jtrix.{base,facet}.*`).

The shared namespace should be fairly small as it expected to be uniform across nodes for a particular Jtrix version.

Additionally, each netlet has a private namespace which is used for packages not found in the global namespace. Figure 18 outlines the namespaces for a node with two netlets.

Netlet namespaces may overlap outside the global namespace (i.e. have classes with the same name). This is instrumental for allowing services to communicate with their clients.

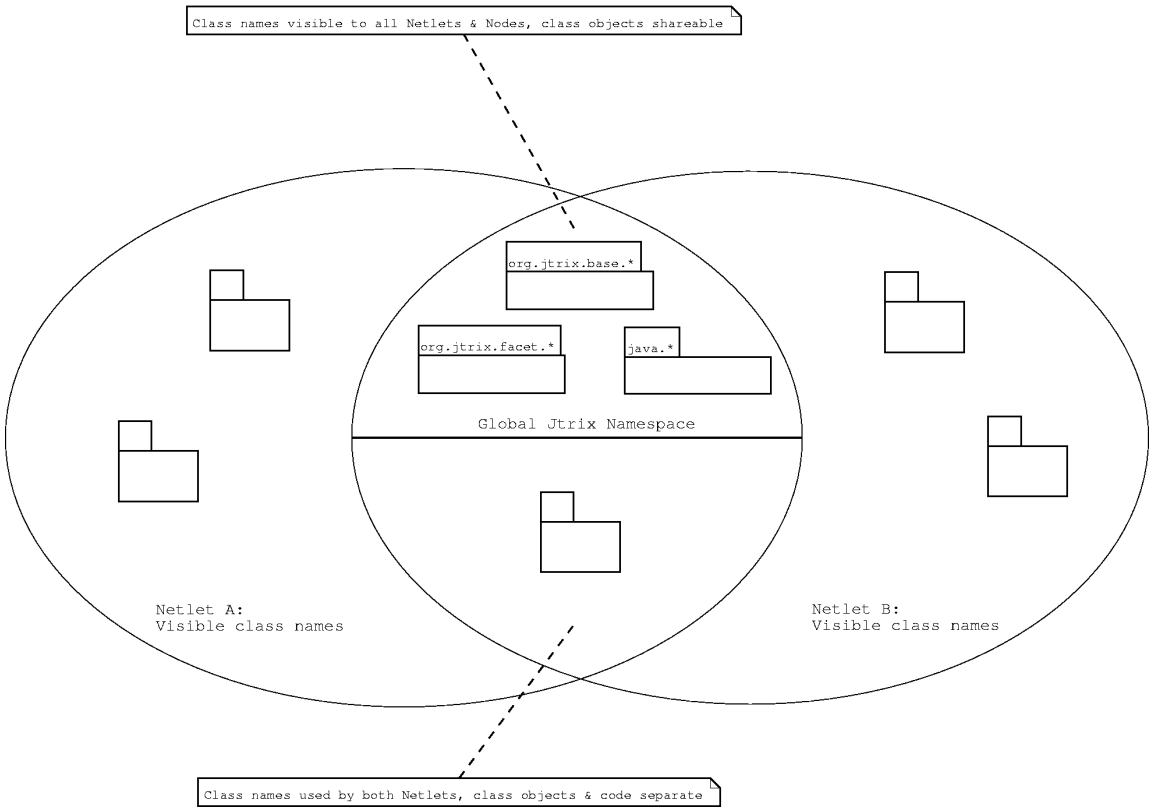


*Figure 18: Netlet namespace separation*

### 8.3.2 Class implementations

Class implementations are not shared between netlets even if they are on the same node. However, a node may allow class implementations to be shared if this is safe and transparent.

## 8.4 Mediation of inter-netlet communication

### 8.4.1 Motivation

- A node must be able to remove any netlet without damaging other netlets. Hence threads must not cross node boundaries.
- All memory allocated by netlets must be "owned" by a specific netlet. Hence netlets cannot share netlet-created objects.
- Netlets cannot export class implementations into the shared namespace as this would lead to naming conflicts.

### 8.4.2 Semantics

A service connection, i.e. the binding of a pair of session interfaces (`IService` or `INode/INetlet`), establishes a mediated communication session between two netlets (A and B). The session is terminated when either netlet dies or calls `terminate()`. Initially, the session contains two mediated interfaces. As described below, new mediated interfaces can be added during the session.

Inter-netlet communication does not rely on a shared class loader for interfaces or argument types. Rather, the mediator checks for structural equivalence between classes bearing the same name.

**Invocation semantics** When netlet A invokes a method on an interface exported by netlet B, either the method returns or an exception is thrown. The passing of arguments and return values follows the Java RMI semantics[8]:

- Primitive types are copied.
- Objects that implement `org.jtrix.service.IRemote` are proxied. By default, the proxy has the type declared in the signature. Using `org.jtrix.base.FacetHandle`, the proxy type can be controlled at run-time.
- Objects that do not implement `IRemote` but implement `java.io.Serializable` are transmitted in serialised form. Serialization requires the Class of the passed implementation object to be the same in both parties' codebases.
- Objects of other types cannot be passed between netlets.
- Exceptions are always checked for equality with the exception classes declared in the signature. If an undeclared exception is thrown (even if it is a subclass of a declared exception), `org.jtrix.base.InternalError` is thrown instead.

As an optimisation, instances of `org.jtrix.base.Immutable` that are held by the node can be passed directly.

The node may limit the number of active invocations into a netlet or into the node.

Mediation introduces new error conditions. A new exception type is defined to indicate those:

```
org.jtrix.base.MediationError
```

**Proxy interfaces** A mediation proxy implements all the application interfaces that were declared by the signature of the method that returned the implementation object. Additional interfaces implemented by the implementation object are ignored. Thus, the type of the proxy is effectively choosen at compile-time.

As an exception, returns of type `IRemote` can be explictly typed at run-time (using the `FacetHandle` class).

In addition, any proxy also implements additional interfaces for controlling the proxy, and for asynchronous invocation.

**Proxy equality** Proxies can be tested whether they represent the same, remote object (`isSameAs()`). Proxies are functionally transparent with respect to equality, i.e. `equals(Object)` and `hashCode()` behaves the same whether applied to proxies and their underlying representation.

However, if the proxy connection breaks, both methods will become unusable (i.e. throw exceptions). Hence, proxy objects should not be stored in containers that continuously use those methods (e.g. `java.util.HashMap`).

The following optimisations are possible without affecting semantics:

[8] http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html

1. the result of `hashCode()` can be cached if the implementation object does not override `Object.hashCode()`.
2. `equals(Object)` can be reduced to `isSameAs()` on the proxies if the implementation object does not override `Object.equals(Object)`.

**Disconnected proxies** A proxy can become disconnected if:

- the underlying implementation object is de-allocated.
- the connection to the implementing Netlet/Node is broken.

In this case, invocations of the application interfaces on the proxy will fail. However, a proxy can be tested at any time to see whether it is still connected.

**Re-exported facets** A netlet can re-export an imported service facet. In this case, the node can minimise the number of mediation levels between end-consumer and end-provider. When re-exporting a facet, the system creates a dependency between the re-exported mediator and the exporting mediator. If the exporting mediator terminates, the re-exported mediator is also shut down.

#### 8.4.3 Asynchronous implementation

**Client** A mediated interface can be invoked without blocking the calling thread. This is achieved through `IAsynchronous.IClient` which is implemented by all mediated objects.

**Server** A mediated interface can be implemented without tieing up a thread for the whole duration of the call. This is achieved by implementing the `IAsynchronous.IServer` interface.

### 8.5 Caveats

- Finalisers from different netlets run on the garbage collector thread. This can be exploited for denial-of-service attacks. Also, it is unclear whether a netlet can always pay for its finalisers.
- It would be useful to dynamically extend a netlet's codebase as in RMI, although that would be a big security hole.

## 9 Versioning

### 9.1 Java version

A Jtrix node is deployed on a particular version of the Java Platform. Equally, it is conceivable for a node to run multiple Java versions at the same time.

A node must refuse to run netlets with unsupported Java or Jtrix versions.

Over time, a node can change the Java Version(s) it supports. If support for a particular Java version is withdrawn by the node, netlets using that versions are required to terminate themselves.

### 9.2 Jtrix version

Jtrix uses a `<major>.<minor>` version numbering scheme. Nodes can be queried for supported versions, descriptors contain a choice of required versions.

A node can change the Jtrix Version(s) it supports. If support for a particular Jtrix version is withdrawn by the node, netlets using that versions are required to terminate.

### 9.3 Netlet version

Netlets are not explicitly versioned.

### 9.4 Code version

Code is versioned using the standard tags in the Jar's manifest[9]. Debugging/diagnostics tools should have access to this information.

## 10 Resource control

### 10.1 Lifetime

All node-issued netlet resources revert back to the node when the netlet dies.

### 10.2 Metering

For each metered netlet (some may be unmetered at the node's discretion), a node maintains a set of resource counters:

- CPU (in milliseconds)
- Heap Memory (in byte-seconds)
- Java object allocations (count)

These counters are visible to the node, and may be re-exported through node facets to privileged netlets.

Notes:

- Counter-based events have been deliberately omitted from the node spec.
- The set of counters may change during the live-time of a netlet.

### 10.3 Role of accounting group

All node resources consumed by the members of a accounting group are charged for together, although a more detailed breakdown may be provided.

[9] `http://java.sun.com/j2se/1.3/docs/guide/versioning/spec/VersioningSpecification.html`

## 10.4 Pricing and charging

Nodes may provide pricing information through the hosting service.

The charging mechanism depends on the hosting contract, and is therefore beyond the scope of this specification.

## 10.5 Non-performance

If adequate and timely payment is not rendered under the hosting contract, the node may shut down all netlets in the associated accounting group.

Higher-level mechanisms need to be employed if the node over-charge or fails to provide resources that have been paid for.

# 11 Interchange Formats

Jtrix uses XML representations for warrants and descriptors. Nodes must accept the following warrant and descriptor formats.

## 11.1 DTD

```
<!-- embedded in org.jtrix.project.libjtrix.warrant.JtrixDTD -->

<!-- shared declarations -->
<!ELEMENT url (#PCDATA)>
<!ELEMENT content-signature (dn,sig)>
<!ELEMENT dn (ne+)>
<!ELEMENT ne (#PCDATA)>
<!ATTLIST ne name CDATA #REQUIRED>
<!ELEMENT sig (#PCDATA)>
<!ATTLIST sig algo CDATA #REQUIRED encoding (base64) #REQUIRED>
<!ELEMENT bind-parameters (param+)>
<!ELEMENT param (#PCDATA)>
<!ATTLIST param name CDATA #REQUIRED>
<!ELEMENT service (dn,public-key)>
<!ATTLIST public-key type (x509) #REQUIRED encoding (base64) #REQUIRED>
<!ELEMENT public-key (#PCDATA)>

<!-- DTD for Warrant -->
<!ELEMENT warrant (warrant-content,content-signature?)>
<!ATTLIST warrant version CDATA #REQUIRED>
<!ELEMENT warrant-content (service?,ap?,warrant-data)>
<!ATTLIST warrant-content serial CDATA #REQUIRED starts CDATA #IMPLIED expires CDATA #REQUIRED>
<!ELEMENT ap (bind-server*|netlet-descriptor)>
<!ATTLIST ap ac-group (internal|external) #IMPLIED>
<!ELEMENT bind-server (url+,bind-parameters?)>
<!ELEMENT warrant-data (#PCDATA)>

<!-- DTD for Descriptor -->
<!ELEMENT netlet-descriptor (descriptor-content,content-signature*,codebase-files?)>
<!ATTLIST netlet-descriptor version CDATA #REQUIRED>
<!ELEMENT descriptor-content (codebase+,secret*)>
<!ATTLIST descriptor-content serial CDATA #REQUIRED starts CDATA #IMPLIED expires CDATA #REQUIRED>
<!ELEMENT platform-version (java-version+,jtrix-version+)>
<!ELEMENT java-version (#PCDATA)>
<!ELEMENT jtrix-version (#PCDATA)   >
<!ELEMENT codebase (platform-version,main-class,parameter-bean?,jar+)>
<!ELEMENT jar (digest,package*,url+)>
<!ATTLIST jar size CDATA #REQUIRED lazy (true|false) #IMPLIED>
<!ELEMENT digest (#PCDATA)>
<!ATTLIST digest encoding (base64) #REQUIRED algo CDATA #REQUIRED>
<!ELEMENT package (#PCDATA)>
<!ELEMENT secret (#PCDATA)>
<!ATTLIST secret encoding (base64) #REQUIRED algo CDATA #REQUIRED name CDATA #REQUIRED>
<!ELEMENT main-class (#PCDATA)>
<!ELEMENT parameter-bean (#PCDATA)>
<!ATTLIST parameter-bean encoding (base64) #REQUIRED>
<!ELEMENT codebase-files (file+)>
<!ELEMENT file (url,file-content)>
<!ELEMENT file-content (#PCDATA)>
<!ATTLIST file-content encoding (base64) #REQUIRED>

<!-- DTD for binding request -->
<!ELEMENT bind-request (url,bind-parameters?,node?)>
<!ELEMENT node (java-version+,jtrix-version+,type)>
<!ELEMENT type (facet*)>
<!ELEMENT facet (#PCDATA)>
<!ELEMENT bind-result (netlet-descriptor,parameter-bean?)>
```

## 11.2 Warrant

Warrants are surrounded by the <warrant> tag. The signature, if present, covers the whole <warrant-content> tag.

Here is an example warrant:

```
<!DOCTYPE warrant PUBLIC "-//jtrix.org//TEXT jtrix-0.1//EN" "http://www.jtrix.org/dtd/jtrix-0.1.dtd">
<warrant version='1.0'>
  <warrant-content serial='0' starts='995375692839' expires='995378032839'>
    <service>
    <dn>
      <ne name='N'>ulf</ne>
    </dn>
      <public-key type="x509" encoding="base64">
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDMrQErOQlAvCerJMFijgcPzKZ7avx6Ae/uD1RA1RHwxqxoT6E5ivjqXRtGXp48YIGsD+IMLg0T
Q6SRj8zK1v+PbttKJ2rsJl6/NuBiZoMlpnkQhChdHyXzodQfYa9HaJTdabVxm5ZDHKMbmJh91XPr1IqBiz1qOKFfbE7DhdNCYwIDAQAB
      </public-key>
    </service>
    <ap ac-group="internal">
      <bind-server>
        <url>http://10.119.8.130:4583/service/1/N%3Dulf</url>
        <url>http://10.119.8.128:4583/service/1/N%3Dulf</url>
     </bind-server>
    </ap>
    <warrant-data>
      <![CDATA[
org.jtrix.project.cluster.comms.AdminRight      ]]>
    </warrant-data>
  </warrant-content>
  <content-signature>
    <dn>
      <ne name='N'>ulf</ne>
    </dn>
    <sig encoding="base64" algo='SHA1withRSA'>
c669hU1JR5T/CEQYOFMfzbumKVLX1XsxkO+plhoC9jnSIo1M1gaqt0Kw3Al7j6ufMFnAI4lThfxNwSkqbbJAjlQi/Hf5Tdgu5t/Rf27vmx/jJboA
j5yUoAKJvO30HZ4tQUAUonj/ttS4tOOeCUxZ9Azjt3vZYbseappM6XUwJg8=
    </sig>
  </content-signature>
</warrant>
```

## 11.3 Descriptor

Netlet descriptors are surrounded by the <netlet-descriptor> tag. The signatures, if present, cover the whole <descriptor-content> tag.

Here is an example descriptor:

```
<!DOCTYPE netlet-descriptor SYSTEM "jtrix.dtd">
<netlet-descriptor version='1.0'>
  <!-- expiry is number of seconds since 01/01/1970 GMT -->
  <descriptor-content serial='349' starts='98438010' expires='98438030'>
    <!-- service ID is optional, but absence may prevent
    Node-wide service registration -->
    <codebase>
      <platform-version>
        <java-version>1.2</java-version>
        <java-version>1.3</java-version>
        <jtrix-version>1</jtrix-version>
      </platform-version>
      <main-class>Main</main-class>
      <parameter-bean encoding="base64">
<!-- serialised parameter bean -->
344394-039504390-adscxbvcre0-igoidf
      </parameter-bean>
      <jar lazy='true' size="2345653">
<digest algo='SHA1' encoding="base64">fde345678901234567890'</digest>
<!-- this is a hint so Node can find classes quickly -->
<package>com.verisign.cert</package>
<package>com.verisign.ca</package>
<!-- JAR sources -->
<url>http://www2.verisign.net/gd/rt.jar</url>
<url>http://www3.verisign.net/gd/rt.jar</url>
      </jar>
      <jar size="54365">
<digest algo='SHA1' encoding="base64">fdegddfcb456346dgdszg0</digest>
<package>org.debian.fun</package>
<package>org.debian.spacemaster</package>
<url>http://www.debian.org/gd/sm.jar'</url>
<url>http://www.uk.debian.org/gd/rt.jar'</url>
      </jar>
    </codebase>
    <secret encoding="base64" algo="RC5" name="tagA">
      308302dadg9431f
    </secret>
  </descriptor-content>
  <!-- signature over descriptor-content -->
  <content-signature>
    <!-- X.500-style distinguished name -->
    <dn>
      <ne name='uid'>prabbit</ne>
      <ne name='ou'>development</ne>
      <ne name='o'>boOthewstreet</ne>
      <ne name='c'>us</ne>
    </dn>
    <sig encoding="base64" algo='SHA1RSA'>
      4909239fs9sd3520534904593089308
    </sig>
  </content-signature>
  <content-signature>
    <!-- X.500-style distinguished name -->
    <dn>
      <ne name='uid'>ulf</ne>
      <ne name='ou'>jtrix</ne>
      <ne name='o'>hyperlink</ne>
      <ne name='c'>uk</ne>
    </dn>
    <sig encoding="base64" algo='SHA1RSA'>
      4909239fs9sd3520534904593089308
    </sig>
  </content-signature>
  <!-- optional, unsigned set of JAR files -->
  <codebase-files>
    <file>
      <!-- the URL ties the file to the descriptor's codebase -->
      <url>http://www2.verisign.net/gd/rt.jar</url>
      <file-content encoding="base64">
        CQENTkFNRToJQ0hSSVMgR1JFU1RZDQ1EQVRFIE9GIEJJUlRIOgkxMCBPQ1RPQkVSIDE5NzANDU5B
        VElPTkFMSVRZOglCUklUSVNIDQ1SRVNJREVTOglTVVJSRVkNDVRSQU5TUE9SVDogCUZVTEwgRFJJ
        VklORyBMSUNFTkNFDQ1wcmVzZW50IFBPU0lUSU9OOglERVZFTE9QTUVOVCBURUFNIExFQURFUg0N
        cmVxdWlyZWQgcG9zaXRpb246CUpBVkEgREVTSUdORVIvREVWRUxPUEVSDQ1wcmVzZW50IFNBTEFS
        WToJozM3LDUwMA0NUkVRVUlSRUQgU0FMQVJZOgmjNTAsMDAwICAhORUdPVElBQkxFIE9OIFJPTEUp
        DQ1ub3RpY2U6CTQgV0VFS1MNCQ0NDVByb2Zlc3Npb25hbCBRdWFsaWZpY2F0aW9ucw1NaWNyb3Nv
```

```
        ZnQgQ2VydGlmaWVkIFNvbHV0aW9ucyBEZXZlbG9wZXIgKE1DU0QpIJcgSnVseSAxOTk5DQ1FZHVj
        YXRpb24gKDE5ODcglyAxOTkzKQ0xOTg3LTg5IJcgNSBBLUxldmVscyCXIE1hdGhzIChBKSwgRnVy
        dGhlciBNYXRocyAoQSksIFBoeXNpY3MgKEEpLCBFY29ub21pY3MgKEIpLCBHZW5lcmFsIFN0dWRp
        ZXMgKEEpDTE5ODktOTIglyBEdXJoYW0gVW5pdmVyc2l0eSCXIEJTYyBIb25zIE1hdGhlbWF0aWNz
        LCBDbGFzcyBJSS9pDTE5OTItOTMglyBTaGVmZmllbGQgVW5pdmVyc2l0eSCXIE1TYyAoRW5nLikg
        Q29udHJvbCBTeXN0ZW1zDSANDQ0NU1VNTUFSWQ0gDSoqIDIgWUVBUlMgQ09NTUVSQ0lBTCBKQVZB
        IERFVkVMT1BNRU5UICoqDQ1DaHJpcyBpcyBzZWVraW5nIGEgcG9zaXRpb24gYXMgYSBKYXZhIERl
        c2lnbmVyL0RldmVsb3BlciB3aXRoIGEgZm9yd2FyZC1sb29raW5nIGNvbXBhbnkuICBIZSBpcyBs
        b29raW5nIHRvIHV0aWxpc2UgdGhlIEphdmEvT08gc2tpbGxzIGhlIGhhcyBidWlsdCB1cCBvdmVy
      </file-content>
    </file>
  </codebase-files>
</netlet-descriptor>
```

# 12 Structure of `org.jtrix`

**`org.jtrix.base`** Jtrix platform API.

**`org.jtrix.facet`** Standardised Jtrix facets for nodes, netlets and services.

**`org.jtrix.facet.node`** Standardised node facets.

**`org.jtrix.facet.netlet`** Standardised netlet facets.

**`org.jtrix.facet.service`** Standardised service facets.

**`org.jtrix.projects`** Projects developed by the Jtrix open source community.

**`org.jtrix.projects.nodality`** A Jtrix node implementation.

**`org.jtrix.projects.libjtrix`** Tools for nodes and netlets.

**`org.jtrix.projects.jtrixd`** A runnable Jtrix node.

**`org.jtrix.projects.cluster`** A hosting cluster.

**`org.jtrix.projects.launcher`** An interactive node for starting and monitoring applications.