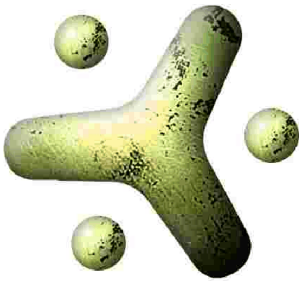


Jtrix: A Technical Overview

$Id: white-paper.lyx,v 1.22 2001/05/11 14:33:11 nik Exp $

Jim Chapman
mailto:feedback@jtrix.org
Jtrix Ltd
57-59 Neal Street, London WC2H 9PJ, UK
+44 20 7395 4980

Contents

List of Figures

1 Introduction

This document introduces Jtrix, a virtual computing platform intended to ease the development and deployment of network-based applications.

1.1 Goals of this document

The goal of this document is to enable the reader to gain a broad understanding of Jtrix: what it is and how it can be used. Jtrix helps solve certain problems—the reader should be able to determine what those problems are, whether his or her problem falls into that class, and how the problem might be approached using Jtrix.

1.2 Scope, intended audience, and related works

This document is intended for a technical audience. A less technical overview is available elsewhere.[1] The properties of Jtrix are described here in a concise and reasonably pedantic manner, but not in intricate detail. Such detail is also available elsewhere.[2]

1.3 How to read this document

The first few sections are fairly independent and can probably be read in any order: Section 2 gives a loose overview of Jtrix, Section 3 details some of the key developments and trends that have influenced it and Section 4 details a sample problem, Harry's trading game, which could be solved with Jtrix.

After that we get to more particular details. Section 5 explains the principles of the software's design and Section 6 describes the resulting system with some specificity. This is followed by the mechanics of writing Jtrix applications, with a particular emphasis on what Jtrix does and doesn't take care of for us (Section 7). One of the things Jtrix does not do is handle distributed application management (though it does facilitate it) so this is covered in Section 8. Finally this is applied to our sample problem described earlier (Section 10) and Section 11 lists some actual implementations of key Jtrix services which, while not part of the core, are important enough to warrant special mention.

2 Jtrix

Jtrix is a reliable, uniformly dynamic, virtual general purpose computing platform which aggregates administratively heterogeneous resources from many sources on the Internet.

Our applications will run on this platform, and use resources which may be discovered and connected dynamically. Our applications will be able to adapt themselves to the changing structure of this platform, and adapt their own deployment and composition to the changing requirements placed on them by their users, or by other applications which in turn rely on them to provide resources.

Here's that first long sentence broken out:

1. Reliable

   The platform allows applications to run with any desired degree of security and reliability. Those using this system to run their applications wish to provide their users in turn with some guarantee of service levels.

[1] Jtrix: An introduction for everyone, available from `http://www.jtrix.org`
[2] Jtrix: Platform specification, available from `http://www.jtrix.org`

2. Uniformly dynamic

   People can contribute resources as and when they have them.

   The platform has uniform properties throughout. Many resources have a localised nature, and are optimally used only by code running on a particular computer. This implies that programs be able to move to properly utilise these resources.

   The platform is dynamic; not constant but continually changing. Resources are continually added and removed. These resources include those resources required to run programs as well has higher level resources.

3. General purpose

   Jtrix is not a specialised transaction or application server. Jtrix is more of an operating system or a distributed component model.

   The operating system aspects allow developers to write to a uniform target platform. This allows code to run anywhere it is required, regardless of the target system.

   The component model allows for reuse. Reuse is possible both for code and for data. Connection to a component may result in the simple download of code, or it might result in a full server/client connection to existing component state. Anything in between is possible.

   Transaction server or content management systems might be built on top of Jtrix, allowing them to pass the benefits to the applications contained within them. Jtrix enables the development of new development abstractions without having to reimplement the boiler plate mechanisms of internet deployment, hosting, etc.

   Jtrix also provides an infrastructure which can enable applications built using these new abstractions to interoperate and share resources.

4. Administratively heterogeneous resources

   The Web has achieved just this in its field of information distribution. Part of the Web's success is due to the lack of red tape involved in publishing a Web site. Given the basic IP connectivity, no more authorisations from any source are required. Web sites under different administrations may freely link to each other. Given basic conformance to the HTTP and HTML standards, no further standards are imposed. Only when the requirements become more stringent do the administrative constraints become correspondingly tighter.

   Jtrix extends the same model to programming and data resources. For any given type of resource, the minimum requirements are very simple; both provider and consumer must adhere to an agreed interface. Pretty much anyone can supply or use such a resource, and resources may be freely linked into consumer applications. As commercial and security constraints become more complicated, correspondingly more complicated mechanisms become involved.

The Jtrix Project is an open source initiative created to promote the use of Jtrix. The project is intended to make large amounts of code available under the LGPL, and to provide facilities and resources for those working on that code. Most importantly, the project provides a forum and facilities for those wishing to promote standards for various aspects of Jtrix.

3 Background

Some of the key developments that have driven the creation of Jtrix:

3.1 Network applications

As the Internet matures, new classes of application are emerging. One interesting development is the network deployment of applications previously regarded as in the domain of the desktop. E-mail, office productivity, and scheduling applications are being deployed on servers using the Web

as the front end. Users gain from the convenience of ubiquitous access to applications and data from any connected device, and from the removal of the need to install and maintain applications on the desktop.

This trend can only continue: hand held devices, Internet connected TVs, and other consumer electronics are becoming more powerful, popular and connected. Server and bandwidth resources become cheaper by the day. The corresponding loss of control of one's own data, and the potential lack of privacy of such a model do not appear to be a concern to most users, or at least do not appear to outweigh the advantages of such applications.

The Web based application model is also attractive because it drastically reduces the amount of state kept on the clients. As long as the user can access a machine with a Web browser the application and the user's data is available.

But just how secure is the user's data? How redundant is the infrastructure providing that application? How can one Web application inter-operate with another? What if the application is withdrawn? Web based applications are subject to the same business pressures as any other Internet site. Many of them are simply user gathering or loyalty capturing systems, intended to draw users to a larger site and subject them to intense advertising. If they cease to perform that service, there is no contract or obligation for them to continue to provide service, or allow users to migrate data contained in them elsewhere.

Current deployment of equipment shows a marked separation between "client" equipment and "server" equipment. Servers are reliable, backed up, and act as central storage and synchronisation points. Clients are unreliable, run applications, and are synchronised with or slaved to servers acting as data repositories. Servers are typically fixed installations, and clients are often portable.

The Web based application is an extension of this model, taking the responsibility for application maintainence and data storage off the home or business desktop or portable device, and moving it onto the fixed Internet or intranet server.

Thin client applications work well when the network resources are available, and less well when the application must function when disconnected from the servers or internet. Traditional applications are still used in such situations.

3.2 Expectation and demand

Internet sites used to be easy to write, because expectations were low and the demand lower. The potential audience was primarily composed of a technically sophisticated minority, interested in the function rather than appearance. Now that wide area networking has gained the public interest, this is no longer the case. Modern applications need much more "gloss" to succeed. Within a fixed development budget, much more time needs to be spent on the finish, often at the expense of underlying technologies for scalability, deployment, intra-application communication, etc.

Jtrix provides a foundation by which many basic technologies may be packaged as components and more freely reused by end user applications. Even such aspects as final deployment and application administration may be packaged in this way. Because Jtrix works at a lower level than content management applications, resources may be shared between applications built using different enabling technologies.

3.3 Internet business models

A big problem for commercial Web sites are the business models forced upon them by the technical architecture of the Internet. Bandwidth and server resources have to be bought, regardless of use. Current e-commerce solutions, which work well for relatively large, discrete purposes such as books, CDs and cars, don't work well for smaller, more frequent transactions. Use of a credit card based e-commerce system isn't workable for a thousand one cent transactions, so sites must either implement accounting systems to batch payments, or give away potentially valuable "product" and balance their books with advertising revenue.

3.4 Scalability and deployment options

A business can either build its own hosting operation, or outsource it. Basic collocation premises provide a secure location, extensible space, redundant high capacity networking, and a guaranteed power supply. Managed server farms extend this concept by providing the server hardware, operating systems and databases. They will make backups and monitor and repair the servers if necessary. Content management operations extend this concept still further by providing a higher level intrastructure in which a network application operates. However, the more technology is outsourced, the more the application or content is forced into conventional modes of operation, as supported by the farm operators. If a business wishes to venture further "out of the box" it must take on more of the responsibility for managing (and purchasing) its own systems. Either way, considerable up-front expenditure is required even for minimal capacity sites: skilled staff, equipment, and infrastructure.

Jtrix tries to extend the "box" into a versatile storage system, which allows the "box" to change shape and size as required.

3.5 The resource barrier

Such a setup leads to a twisted world. To effectively gain advertising revenue, or gain capital investment, one needs to demonstrate a large user base. To get the large user base, one needs a flashy, expensive launch with lots of media hype, and pretty graphics. Valuable resources go towards the effort to push the business over this barrier, instead of going towards the development of a solid application. Businesses need to spend their limited startup resources on this push, rather than application development.

3.6 Innovation

Correspondingly, innovation on the Web seems to have slowed from an end user perspective as these entry level requirements for a public site have been raised. Any site hoping to service a large number of users needs high bandwidth, the high availability provided by multiple servers, multihomed networks, and load balancing systems. These are all out of reach for a small developer or business starting with just a good technical or business idea, discouraging such people from having a go.

For any application, the development vision is likely to be compromised by the business reality of Web deployment. The need for coordinating the development process with the launch publicity results in very hard deadlines, which in turn discourage the use of higher risk technologies, resulting in a tendency to go with tried and true systems and techniques, because they minimise risk, eliminate the need to solve basic deployment details, and can be outsourced.

Indeed, much of the risk associated with new technologies is not associated with the application itself, but in the re-solution of these details. And a truly innovative application may end up stretching an existing platform to or beyond its limits.

If "tried and true" technology existed at a lower, more general level, it could be used to support existing high level abstractions, and still be compatible with new ways of doing things.

The Jtrix model allows functions to be outsourced from the application development process at a much lower level, and in a much less contraining way than with high level Content Management or Application server systems. It allows a developers "tool kit" to provide the basic functions with less constraining baggage associated with them.

3.7 Portals

Portals have become common "sights" on the Internet and related domains. Portals provide a service which is aggregated from many different sources. For instance, a shopping portal allows

different vendors to advertise and sell their wares through a single site. The buyer can select and purchase goods using a single shopping basket and payment for all vendors.

From a Jtrix perspective, this could work in two ways. Both approaches may be combined freely to find the best solution.

The first approach is for each vendor to be treated as a resource by the portal. The portal is given the "right," under contract, to sell goods provided by the vendor. The service provided by the vendor to the portal includes the ability to inspect inventory, place orders, and track delivery status.

The second approach, subtly different, is for the portal to be treated as a resource by the vendor. The vendor is given the "right," under contract, to advertise goods through the portal. The basic services provided by the interface are the same, the difference is the party driving the formation of the contract.

Implementing the full portal interface may be quite complicated. Shipping, inventory tracking, e-commerce, and such all have "offline" aspects to their implementations. Jtrix resources are dynamic and may be aggregated. Vendors (or portals) may subcontract some of the functionality from more specialised services in order to provide the full portal interface. This may be done with varying degrees of transparency to the end user of the service.

Consider a "wholesaler" who maintains a small inventory, requesting more stock as required, and handles delivery to purchasers. Such a service might be sub contracted by vendors who wish to supply the whole portal interface. This subcontract would be invisible to the portals with which the vendor deals. Alternatively, the "wholesaler" might act as an aggregated vendor, making its own deals with portals and providing a value added service.

In all cases there exists a contractual framework to define the responsibilities and risks incurred by all parties.

4 Harry's trading game

This scenario is an abbreviated version of the scenario in the introductory description of Jtrix, so if you've read that document, you can skip this section. It reiterates the problems expressed above using a hypothetical business startup scenario.

Harry, our representative developer, could be creating a Web site, starting up as an ASP, becoming an ISP or any one of a hundred other things. He happens to be creating an on-line game, but his problems are universal...

4.1 Harry's game

Harry is a student in his final year of college. He has dreamt up, but not yet implemented, a vast on-line trading game which could make his fortune and perhaps even cover his college fees. People will interact with each other and with pre-programmed robots; they will buy and sell resources within the game, and boost themselves with top-ups of real money. And they may even win real money, too. It will be played on home PCs, on games consoles, palm-tops and mobile phones. They will play it from the USA to Japan, Sweden to South Africa. It will start off small, but it will sweep the world. And it will all be one single all-pervasive game. (See Figure 1.)

Harry has the development skills to build the core application, but beyond this it will involve:

- Basic resources. Disk space, CPU time, bandwidth, memory, IP addresses.
- A central database. For the players' basic information.
- An application front-end. In many varieties, for PCs, games consoles, and more.

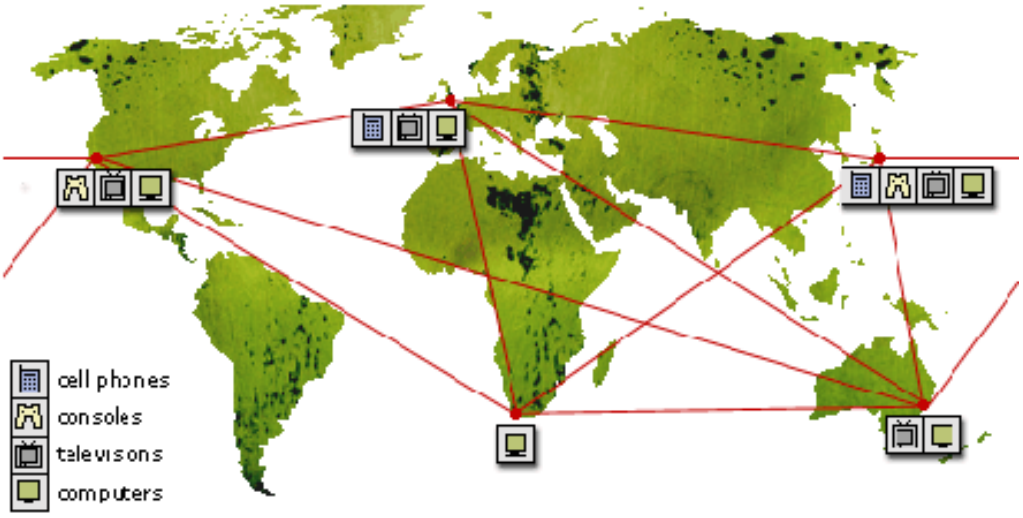


Figure 1: Harry's application will need different resources in different instances. Games consoles may be a significant platform in Japan long before they become popular in South Africa.

- Mobile comms front-end. He'll need to gateway into cell phones. And whatever the next technology is.
- Third party services. Mailing, chat facilities and other services will be needed sooner or later.

That will do for starters. Now Harry needs a technology base.

4.2 What Harry needs

Easier said than done. In his design and planning Harry needs to consider a few things. He needs. . .

- To make money quickly. Harry cannot afford large up-front costs, which he thinks is quite reasonable since resource usage will be low in the early days.
- Smooth growth and evolution. Harry cannot afford to rebuild his infrastructure every six months as demand grows.
- Managed services. He will look after the application-level, but someone more used to systems administration can worry about the low level details: disk space, uptime, backup, and so on.
- Flexible hosting. As demand fluctuates the hosted services will have to move closer to the players. If Japan takes off he needs to put servers there fast. He cannot realistically plan three months ahead.
- Scalable applications. His applications need to spread themselves out across several locations to cope with expected load. Harry does not want to implement each new setup again and again.
- More resources. Harry will want to expand his database, disk space and so on very quickly. He should be able to do this dynamically.
- Reliability. When Harry's game gets big, he doesn't want the whole thing to drop dead just because of a single failure. This can be expensive: either building it to be reliable to start with, or rebuilding when he can afford the extra hardware. Harry wants to build in reliability cheaply from the beginning, and then spend money only on the extra resources needed to deliver it.

- Adaptive applications. Harry's software must be able to upgrade and improve itself. If the game is used heavily in Australia then the database needs to start caching results locally. It must be able to intelligently add or drop functionality as it sees fit for that particular situation.
- Flexible applications. An application should be able to choose the right version of itself for the right environment. On a palm-top an application may have more intelligence but less connectivity than than the same application on a mobile phone. Some PCs have more disk space than others. Any application should be able to choose the right version of itself for that particular environment.
- Choice of vendors. Harry's applications must be able to switch service vendors (for storage, messaging gateways, hosting, etc) at will. He may find a better deal for a service, possibly just for one geographic location. There must always be choice, and a choice at design time is not good enough.
- Reasonable ASPs. Harry wants to find ASPs who can deliver exactly what he wants and no more. He does not want to have to pay for bundled services he will not use.
- Realistic costs. Harry wants to pay for the resources and services he uses. He does not want to have to pay large set up costs to ISPs and ASPs.
- Bank integration. Money will have to change hands, and will have to change hands between several banks. Harry does not want to have to sort that out. The banks can sort it out between themselves.

4.3 Harry's choices

When Harry looks at what he can use, his choices do not look good. . .

- Distributed computing platforms. There are several of these available. But deployment with these is difficult. Harry's code needs to move around. It needs to find new places to host itself as it grows to meet players' demands. And it needs to do this by itself. Current platforms can't do that. They need a bit of help.
- His own hosting. Harry could become his own ISP and run his own trusted global network. But this is not his core skillset. He also finds himself several million dollars short.
- Traditional ISPs. Leave it to the experts. The downsides are: large up-front costs; global integration is difficult and expensive (particularly when he will be using different ISPs); and if he wants to set up in Japan then he needs to start spending several months ahead, with a prediction that that market will take off when he is complete—and he cannot afford to be wrong.
- Write everything himself. Harry will write the basic application, but does not want to re-invent services which others already provide.
- Traditional ASPs. He can use ASPs, but this is usually a design-time decision. Changing to another ASP will be a pain. Harry does not like vendor lock-in. And he will spend too much time integrating with them.
- Remote-services platform. Some services are provided remotely. They use protocols like SOAP. But not every service is suited to this. And even if a particular service usually is, then that will still not be right for every client machine. Not everyone is always connected. An alternative is. . .
- Application-oriented systems. But again, not every service and every players' machine is suited to large client-side applications. The third party services he uses—and his own—must be adaptive.

Figure 2: Harry's game needs to evolve smoothly and efficiently as popularity of the game and the various platforms change. Mobile telephony bases need to move to the right location to suit changing demand; games console usage will not grow predictably, but he still needs to be responsive.

- Money. Harry's game code should pay its way, but he cannot afford large up-front costs. He can afford to pay an ISP if his game uses a little bandwidth, but he does not want to pay the ISP's minimum charge of a 2Mb link on day one. He does not want to buy 100 servers months before he needs them. VC funding would help him in the short term, but would reduce his share of the profits in the end. Harry wants to pay for what he uses, and no more.
- Building in bank integration. As his application moves and grows round the world, so the players will want to pay from various bank accounts in various currencies. His code needs to enable it, but the banks should be able to work out the detailed mechanics themselves. That's a quagmire for someone like Harry.

4.4 Harry's problem in a nutshell

Harry's problem is this: the network is passive.

When we say "network" we mean not just the wires between servers, but also the servers themselves and all the resources and services they offer. It is this power-base which is passive.

Harry can write applications that are network-aware, but he needs the network to be application-aware.

He needs service providers—ISPs and ASPs—to offer services to his applications. He needs his applications to be able to spread themselves out to new, better locations, with the right resources, and continue their service. He needs the network to tell the applications what kind of appearance is best for any one location. And then his applications can make intelligent choices. He needs the network to help his applications in an organic way. (See Figure 2.)

If Jtrix didn't exist, he'd have to invent it.

5 Design Principles

Some key concepts which run through Jtrix's design:

5.1 The end-to-end argument

This is nicely elucidated elsewhere.[3] To summarise, the end-to-end principle argues that we should not build unnecessary complexity into our base system because at a general purpose level of abstraction we can't hope to properly solve problems which exist at more specific levels.

5.2 A simple base layer

Jtrix's take on the end-to-end argument is that implies that the essentials be factored out into a base abstraction layer. This layer, which must be kept as simple as possible may then be well understood and more easily implemented.

We call this layer the "Jtrix base API" and it forms the common basis by which applications and their resources communicate, and provides the homogeneous runtime environment for Jtrix applications.

The Jtrix base layer does not directly implement all the properties we desire, but enables applications and supporting services to implement them.

Most Jtrix applications will be multi-tiered, using support functions built in as lower tiers but above the base API. Such tiers may be independently versioned, running using the same base API. Existing programming abstractions such as CORBA and high level transaction servers or content management systems might form these higher tiers.

5.3 Minimisation of central authorities

One of our principles is to avoid creating central authorities. Many problems are more easily solved when one postulates some central service that can coordinate and authenticate an effort. Jtrix applications are characterised by organic, dynamic deployments and linkages between disparate entities.

If an application needs a central authority, we should at least allow people to choose it. And give them a real choice, rather than a simple blanket statement like "Well, you can use any authority you like, but if you don't choose this one, it won't work."

Central authorities, while being generally useful, present problems. They can increase the amount of red tape involved and violate our wishes for the system to enable the use of administratively heterogeneous resources. Indeed, one could probably define "administratively heterogenous" (in our introductory sentence of Section 2) as the absence of a central authority.

5.4 Security issues

Our base API supports the minimum functionality which will allow our applications to make themselves secure and reliable.

It's a general truth that security and reliability come with an associated cost. Different applications have different criteria, and the end-to-end argument applies. We don't want to impose high security on applictions that don't need it (and don't want to pay for it). Conversely, we don't want to prevent applications from implementing it, or impose lower security standards than required.

Security generally involves the following issues:

1. Application integrity

   We don't want unauthorised parties to be able to look at or affect our code and data. We need to be able to control the type of system that is used to run our applications.

[3] `http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.txt`

2. Performance and reliability

   We are classing reliability and performance in with security because the same principles apply to both aspects of a system. Applications provide services, and continued and timely provision of that service may be compromised either by security breaches or by failure. Reliability of an application may well affect the security of another.

3. Auditing to detect compromise and provide data for correction or punishment

   In the event of a security violation, we want to know that it happened, and we want to know what happened, and how. This may enable us to prevent it happening in the future, repair the effects of the violation, and find the responsible party.

4. Paying for the damage done

   Security violations cost money. The waste of time spent correcting them, the costs associated with loss of data, reparations to those who have been damaged in turn by the violation.

5. Control of risk

   Insecure and unreliable resources will exist, and may not be suitable for applications with security constraints. A key feature of Jtrix must be to allow applications to control their exposure to risk. Applications can determine the potential exposure which will arise from the use of a particular resource, and are able to avoid using it or adapt their operation to enable the safe use of that resource.

6. Hosting security

   This applies in particular to hosting resources. A program running on a machine can be only as secure as the machine itself. The potential exposure can be evaluated before anything is deployed to the host in question. Depending on the results of the evaluation, code may not be transmitted, or its function may be changed to suit the new environment. An insecure platform may be quite usable for some applications, but not for others, at least without special precautions.

5.5 Encourage specialised development

People have strengths and weaknesses. The system must allow someone to concentrate on their strengths, and outsource the problems they do not which to address. This must be done with a fine enough grain so that the outsourced functions need not create more problems than it solves.

5.6 Contracts

Many functions will be outsourced. Applications use services from other applications. Such interactions must take place using a defined pattern of responsibility, performance guarantees, and liability for non-performance. It must be possible to detect and demonstrate non-performance of a contract.

5.7 Payment

When using a resource, choice promotes competition, and competition reduces cost. It should always be possible to discontinue the use of a resource in favour of a competitor. Such a move, as in the physical world, will carry an associated cost-to-change.

Ad hoc resource use and dynamic deployment encourage applications that can react quickly to changing resource availability. Use of a particular resource, or a particular deployment may represent optimal performance at a certain time, but changing circumstances may indicate redeployment or a switch of resource supplier at any time.

This seems to imply that pay-as-you-go type services are to be preferred over services which are paid for using long term contracts. Jtrix needs to support payment mechanisms natively, such that payment for service may be made alongside use of the service.

The end-to-end arguments apply. Any payment mechanism we build into the base API will likely be inadequate. Payment mechanisms are a potential minefield of security, liability, banking laws, currency conversion and exchange rates, inflation, tax, and legacy systems. The base API must provide an enabling technology, allowing payment to be transferred in a standard manner, but at the same time dictate almost nothing about the mechanisms involved.

5.8 Component architecture

In order to get effective choice of a resource, different implementations must conform to some standard. Component based development, where a resource becomes a black box component with a defined interface, is a standard tool in modern programming. Components provide resources in Jtrix, allowing applications to treat resources as black boxes with defined, standardised service interfaces.

The base API provides a component model which reflects the uniformity requirements of the platform, but allows the components to control their exposure to risk. Interactions between components take place according to defined contracts and it is possible to audit and enforce the performance of those contracts.

A component architecture, plus the ability to switch suppliers and to pay-as-one-goes is useful in the creation of niches where specialists can flourish. This model creates a bazaar effect, where components can be built from other components, providing a uniform or value added service from some number of more basic resources.

5.9 Versioning

Jtrix itself can be regarded as a central authority, with respect to the versions of the base layer. While there's no way around this, as one needs standards in order to communicate, this requirement confirms our philosophy that the Jtrix base API should be as small as possible, and openly available. Minimising the base API means that it will change less often. Each incompatible version of the base API effectively partitions our virtual platform and is to be avoided.

The component model provides a mechanism whereby components can be upgraded, and new or improved services supplied while maintaining backwards compatibility for older clients.

5.10 Legacy systems and systems integration

Jtrix is not going to take over the world. It must be possible to represent legacy systems and other non-Jtrix functionality as Jtrix resources. In addition, it should be possible to allow non-Jtrix applications to use Jtrix functionality.

External integration is particularly important in the portal domain. Portals provide an end user service which aggregates and adds value to services provided by other suppliers. Jtrix's component model, with its dynamic security and its basis on contracts is an obvious win for this type of application. While we might feel that it would be better to implement portal style applications directly in Jtrix, using the component model for portal-vendor integration, a more likely scenario is that existing portal code might want to incorporate services provided through Jtrix. By encouraging the integration of Jtrix into external programs, we can allow this kind of scenario.

In addition, it's important to be able to use existing code inside the Jtrix environment. Support should be retained for familiar ways of doing things, such as accessing files or using network sockets.

6 Jtrix defined

The base Jtrix API is strictly defined in the platform specification.[4] This section gives an overview of the facilities provided, partly to enable the reader to gain a quick understanding, but primarily for discussion purposes. For exact details, see the specification.

6.1 Glossary

A *node* is the physical runtime platform for a Jtrix application. A node may span one or more machines, or (probably more for demonstration purposes and testing) several nodes may run on one machine. Several Jtrix applications can run simultaneously on one node.

A *netlet* is a self contained program which runs on a Jtrix node. A netlet is written in terms of the Jtrix base API, which is implemented by the node. A netlet runs in a virtual environment, allowing platform mechanisms such as files and sockets to be used, even though these mechanisms may manipulate simulated or remote resources.

A *service* is a resource available to a netlet.

A netlet is entitled to use any service for which it holds an appropriate *warrant*. A warrant is evidence that a contract exists between the netlet and the service. Warrants may be transferred between netlets, after which the recipient netlet may use the service. Different warrants may apply to the same service, conferring different rights to that service.

A netlet is said to *bind* a warrant when it connects to a service. When a netlet binds a warrant to connect to a service, an appropriate new netlet is created on the node to supply the service. The netlet created may be specified statically as part of the warrant or dynamically by a binding server. The two netlets are then connected.

The connection between the two netlets is the *service session*. A service netlet may, under certain circumstances, supply the same service to more than one netlet. In this case we get a second service session.

Through a service session, a netlet may access one or more *facets* which are interfaces implemented by the netlet providing the service. These facets provide the service functionality. Depending on the warrant used to bind the service, different facets may be available.

All these terms are discussed in more detail in the following sections.

6.2 Nodes and netlets

Netlets are the smallest programming unit directly addressed by Jtrix. As indicated above, a node provides the environment in which a netlet is executed.

Currently Jtrix netlets must be written in Java^{TM}, but other languages may be supported in the future.[5] Java satisfies the portability requirements, allows Jtrix to be implemented on many operating systems, and is easily incorporated into other applications.

A netlet consists of one or more Jar files which contain its programming. A netlet has a main class of which a single instance is created by the node when the netlet starts. The node controls the netlet though this object, which must implement the netlet control interface (INetlet) specified in the Jtrix API. When the netlet is initialised, it is passed an instance of a node control interface (INode) through which it accesses functionality provided by the node.

[4] Jtrix: Platform specification, available from `http://www.jtrix.org`

[5] Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

6.2.1 Descriptors

A *descriptor* is an XML document which describes to a node how to create a particular netlet. The descriptor may be digitally signed by the netlet creator.

The descriptor contains (directly or by reference) the Jar files comprising the netlet, the name of the main class, and arguments which are used to configure the new netlet. It also contains a secret which enables auditing and status information concerning the netlet to be yielded to any holder of that secret.

Multiple versions of the Jtrix base API may be supported by one descriptor. All programming-related data in the descriptor may be API version specific, and more than one API version may be supported.

A descriptor may be contained in a warrant. In this case the warrant shows not just the right to use a service, but also the specific netlet used to provide access to it. However this carries trust-related penalties—see Sections 6.2.5 and 6.3.6.

6.2.2 Arguments

A netlet may have two arguments, one of which is contained in the descriptor, and one which is passed alongside the descriptor. These are the “trusted” and “untrusted” arguments respectively. Both are passed to the netlet on initialisation.

6.2.3 Hosting services

The only mechanism directly defined in the Jtrix API for the execution of a netlet is the service binding mechanism. Additionally, netlets may also be executed through higher level services provided by the operator of the node. The mechanism by which these services execute netlets is a property of the particular node implementation. A service which provides for the execution of netlets is deemed a *hosting service*.

6.2.4 Principals

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. Any principal will have exactly one name.

6.2.5 Trust relationships

A netlet which is signed by a certain principal may be given privileges by a particular node and which are not available to unsigned netlets. These privileges may allow the netlet to perform administrative functions for the node, either directly through the native language API, or indirectly through the node-netlet control interfaces.

Before a descriptor is passed to a node in response to a binding request, the binding server may obtain signed certificates from the node as part of the binding mechanism. Services which must apply particular constraints to the nodes upon which they execute may use these certificates to ensure that the node meets these criteria. Depending on the certificates held by the node, the service may respond to the binding request with different descriptors, or may refuse the binding request.

The opportunity for this evaluation is not present when the warrant for a service contains the netlet descriptor, bypassing the binding protocol. Thus this technique must be used with care to ensure that security is not compromised.

The base API provides facilities for one netlet to access the descriptor signatures of a netlet with which it has a service connection. Clients may access the signatures of servers, and vice versa.

6.2.6 Accounting groups

Netlets are grouped within a node into *accounting groups*. All resources consumed by netlets are billed to the accounting group, and each netlet belongs to exactly one accounting group. The mechanism by which bills are calculated and paid is a property of the contract between the netlet owner and the operators of the node.

When a netlet is executed through a hosting service, the contract is determined by the warrant used to access the hosting service. When a netlet is executed to satisfy a binding request, there are two possibilities:

1. The new netlet becomes part of the accounting group of the requesting netlet.

   The netlet binding the service is thus responsible for all charges incurred by the local provision of that service.

2. The new netlet joins a newly created accounting group.

   The new netlet is thus liable for its own resources. The contract here is determined through the binding protocol.

The warrant for the service may specify the way in which the new netlet is created.

6.2.7 Node resources

Local node resources are not necessarily available to netlets. Services that provide disk, network, and other resources local to a node may be accessed through the normal language APIs through a trampoline mechanism, allowing code written against the normal language APIs for disk I/O and networking to be re-used. The trampoline mechanism is used to make any compatible service available through the normal APIs and does not imply that the resources are being supplied through the local node. The API supports trampolining to the extent made possible by the language.

Basic networking facilities must be available to all netlets, and are billed according to the netlet's contract with the operator of the node.

6.2.8 Netlet lifetime

The lifetime of a netlet is limited by the lifetime of a particular node. There is no direct provision for a netlet to transcend that lifetime. The contract under which a node executes a netlet, or the certificates associated with the node may imply certain guarantees about the node's reliability.

Hosting services may offer guarantees over and above this, providing facilities by which applications may transcend the lifetime of any particular node.

6.2.9 Isolation

Netlets running on the same node may not communicate with other netlets except through services. The contract under which a netlet runs on the node indicates the privacy which is guaranteed to that netlet.

6.2.10 Netlet/node interface

The netlet interface implemented by every netlet provides facilities for the node to:

- Initialise the netlet.

  Netlets are initialised after construction. The initialisation method passes the netlet the controlling node interface, and the arguments with which the netlet is being run. The netlet may bind services and use other node facilites during its initialisation.

- Terminate the netlet.

  The netlet should terminate itself in a timely manner. It may be terminated abruptly at the discretion of the node. When this request returns, the netlet will be considered terminated. All service sessions will be shut down, any threads terminated, and all resources deallocated.

- Create a service session from the netlet.

  In response to a bind request for which this netlet is the server. This creates a service session.

- Request a netlet facet (see Section 6.3.11)

- Request a list of supported netlet facets (see Section 6.3.11)

The node interface provided to each netlet provides facilities for the netlet to:

- Indicate that it has terminated.

  The node will use the above termination mechanism to terminate the netlet.

- Request the binding of a warrant.

  The warrant will be bound and a service session created.

- Request a node facet (see Section 6.3.11)

- Request a list of supported node facets (see Section 6.3.11)

6.3 Services, contracts and warrants

A service provides services under contract. The contract itself is not addressed directly by the Jtrix base API—it may be of any form, including off-line. The API deals with warrants, which are evidence that a contract exists for the service.

6.3.1 Service state

A service may maintain state information. The conditions under which the state is maintained, and the availability of the state information, is covered by the contract under which the service operates.

6.3.2 Service identifier

A service may have an identifier. The identifer consists of a public cryptographic key and an X.500 name. Since a service contract corresponds to exactly one service, a service contract corresponds to at most one service identifier, even though there may be several service contracts corresponding to that service.

6.3.3 Warrants

A service warrant contains:

- An optional service identifier corresponding to the provider of the service.
- A data block used only by the service and its netlets. This data block is opaque to the node.
- Binding information or a descriptor.
- An optional accounting type, one of the following:

  Internal The new netlet is to join the requestor's accounting group
  External The new netlet must have its own accounting group

A warrant may be digitally signed by the service provider—i.e. its principal.

6.3.4 Binding: The process

When a warrant is bound, a descriptor will be determined for the netlet which is to provide the service. If the warrant contains a service identifier, then the descriptor *must* be signed by the service provider.

The descriptor is determined in one of three ways:

1. If there is a netlet already running which may be used to provide the service, that netlet will be used.
2. The warrant may contain the descriptor.
3. The dynamic binding process is used.

These process are discussed in the next three sections.

6.3.5 Binding via service netlet reuse

When a warrant is bound by a netlet, the node may use an already existing netlet to provide the service. A netlet may be reused in such a way only if certain security and financial criteria are met:

- Security criteria

  These criteria ensure that the existing netlet really does implement the intended service, preventing third parties masquerading as the service provider. They must all hold:

  - The warrant being used to bind the service contains a service identifier.
  - The warrant is signed by the provider.
  - The descriptor from which the existing netlet was created is signed by the same provider.
  - The warrants are identical.

- Accounting criteria

  These criteria ensure that one client does not pay for the provision of a service to an unrelated client. This might be the case if a single service provides for multiple, unrelated contracts. Either may hold:

  - The existing netlet was bound into a specially created accounting group whose resources are being paid for by the service provider.

    -or-
  - The existing netlet is in the same accounting group as the netlet performing the bind.

The node may choose not to re-use service netlets. A service must not assume that a netlet will be reused.

6.3.6 Binding via a descriptor embedded in a warrant

The descriptor may be contained in the warrant. This provides a convenient mechanism for “quick and dirty” services. There are some limitations to this mechanism of which the client must be aware:

- There is no provision for evaluating the node before execution.

  The descriptor and any data it contains are available to the node before it can be determined whether the node is trustworthy. The method should not be used for any purpose requiring real security.

- There is no provision for the negotiation of a contract with the node.

  The base API only allows embedded descriptors to be used to create services in the same accounting group as the client. If the warrant indicates the new netlet is to have a newly created accounting group then the descriptor will be rejected.

6.3.7 Dynamic binding

The most general case for binding is to use a dynamic binding protocol. The warrant contains one or more binding server definitions, each of which contains:

- A list of URLs
- A set of key/value binding parameters

Multiple binding protocols are supported. The base API specifies that every node must support a simple HTTP based binding protocol over normal and SSL based connections.

Binding protocols involve the following steps:

1. The node connects to the binding server with the specified URL.

   The binding parameters associated with the URL are transferred, along with the accounting type if present in the warrant.
2. The node may make available any signed certificates with which it is associated, if the protocol supports it.
3. The binding server may determine the host's capabilities (see Section 6.3.11)
4. The binding server may negotiate a hosting contract if the protocol supports it.

   Payment information and signatures may be exchanged as part of this process. This step may not be performed if the warrant specified an internal netlet.
5. The binding server determines the descriptor and unsigned argument to be used.
6. The descriptor and arguments are transferred to the node.

The simple HTTP binding protocol does not support contract negotation. If the service provider netlet is to pay for its own accounting a standard contract is used with a pay-as-you-go model. Payment must be arranged at binding time, and involves the transfer of a special warrant to a payment service. See Section 6.4.3 for more details of the wallet mechanism.

Node certificates are not transferred unless the SSL transport is being used, in which case the certificates are sent during the SSL negotiation phase.

6.3.8 Service session

When a warrant is bound, a service session is created. The service session involves a bidirectional connection between each of the netlets and the node, and provides facilities for:

- Terminating the session
- Obtaining a list of facets available
- Requesting a particular facet implementation
- Receiving notification that the session has been terminated
- Providing a list of supported facets
- Providing a particular facet implementation

The session is set up as a symmetrical connection, with each netlet providing and receiving an implementation of the base API's service session interface, IService.

6.3.9 Session lifetime

The service session continues until either netlet terminates, or either netlet terminates the session. The session is considered terminated as soon as one of these events occurs. The session interfaces and any facets belonging to the session will be unusable. A netlet may not use the session interface or any of the session's facets even during the notification.

6.3.10 Service Facets

A service is implemented in terms of a set of abstract Java interfaces. These interfaces are called Service Facets and may be requested or provided by either party in a service session. All interfaces have additional properties which are added by the connection process:

- Asynchronous invocation of any of the interfaces methods
- Notification (via callback) of the termination of interface function

The set of facets available from a service may change over time. As service APIs are enhanced, the changes should be reflected by additions to the set of facets available. Backwards compatibility may be maintained by implementing the old facets.

Facets are restricted to data types which are Serialisable, or which are Facets themselves.

6.3.11 Node as a service

The node to netlet connection has similar semantics to a service session. Facets may be exchanged between the node and the netlet, and the node and netlet interfaces are Service Facets. Facets implemented by the Node and Netlet are used to provide various extensions of the basic services provided by a node.

6.4 Standard service definitions

The Jtrix base API defines some standard IRemote interfaces which can be used in conjunction with the base API. These are kept as simple as possible. Sophisticated services may offer more functionality than can be expressed using these interfaces, but should still implement these interfaces to allow use with the base API.

6.4.1 Disk I/O

The disk I/O interfaces implement the semantics necessary to trampoline a storage service through to the language API for disk I/O. In this way, a storage service may be used with existing code that manipulates files on disk.

If no service is trampolined in this way disk I/O facilities will not be available to a netlet.

6.4.2 Networking

The network I/O interfaces implement the semantics necessary to trampoline a storage service through to the language API for networking. In this way, a communication service may be used with existing code that uses networking.

If no service is trampolined in this way, basic networking services must still available.

6.4.3 Wallets

The wallet service is the means by which payments are made. Wallets provide a general payment abstraction without constraining implementation details. Wallets may be used to represent digital cash, credit accounts, bank accounts or any other kind of currency.

Jtrix specifies the basic negotiation interface between wallets. This interface is used to determine a payment interface supported by both parties. The payment interface determines the currencies in use, exchange rates, etc. It's anticipated that standard payment interfaces will arise.

There are two basic patterns supported by the Wallet Interface:

1. Seller Initiated

   The Seller passes a Wallet Warrant (The "Sellers Wallet") to the Buyer. This Wallet acts as a "destination" for payments. The Buyer uses its own Wallet to "push" money to the Seller.

2. Buyer Initiated

   The Buyer passes a Wallet Warrant (The "Buyers Wallet") to the Seller. This Wallet acts as a "source" for payments. The Seller uses its own Wallet to "pull" money from the Buyer.

In either case, both parties have their own Wallet. The party which receives the Wallet from the other initiates the transaction between the Wallets. The party with passes the Wallet gets to veto or inspect the transaction, either when the transaction takes place, or in advance.

The degree of trust between the wallets may affect the efficiency of the transfer, and therefore the cost per transaction and the minimum transaction size. To illustrate this, consider the following scenarios:

1. Buyer B purchases something from seller S. To pay, B uses a wallet service supplied by their bank account with Second Fictional Bank, Inc. The seller has a wallet from First Fictional Bank, Inc. The only interface they have in common is a credit card style debit transaction, and First Fictional Bank has never heard of Second Fictional Bank. So the transaction cost is quite high, because each debit needs to be authorised through the credit card agency.

2. Buyer B and seller S both use wallets from First Fictional, Inc. In this case, the wallets can use their commonality to negotiate an "Intra-First Fictional" payment interface. Payments can be transferred directly with little cost, direct from account to account.

3. Buyer B and seller S have different banks, but both banks use a wallet service from First Fictional Clearing Systems, Inc, which operates a digital cash mechanism. Each wallet leases digital coins from its corresponding bank account. Because the wallets trust each other, digital cash is transferred without verification. Coin copying is ruled out because the implementations and operation of the wallets are known and trusted not to do illegal things.

Other digital cash techniques could be used to aid in the transaction between untrusted parties, and minimising the need for third party verification, thus reducing the transaction cost. Thus the Jtrix wallet system can adapt to improving technology. A technology which is more effective at reducing risk correspondingly reduces the constraints by which wallets need to protect their data from untrusted systems, and make verification transactions.

Wallets really represent a direct communications path between the financial institutions involved. They enable those financial institutions to use smart proxies at the point of sale, if desired. In a cash model, the financial institution degenerates into a real wallet analogue.

6.5 Service payment

There are various models for service payment, and various possibilities for implementing them in Jtrix:

- Pay as you go.

  When a service is bound, a standardised facet is used to transfer a wallet warrant from the client to the server. The standard facet would be implemented *by the client* and requested by the server.

- Pay on contract negotiation.

  A fixed fee is charged. This is paid at contract negotiation time (possibly offline) and time-limited warrants issued for the service. Payment is not required when the service is used.

- Account based payment.

  When the contracts are negotiated, an account is set up. The service issues an extra warrant to allow the account to be inspected or maintained. Payment is not required when the service is used.

- Controllers and users.

  Two kinds of warrant are issued, one for simple users, and one for controllers. Users do not pay, but their operations are reflected to controllers for authorisation and payment collection.

- Service-specific currency

  A service can be paid for using a soft currency mechanism which is managed separately. This could be used by an application to control the usage of the resources for which it is paying by controlling the distribution of this currency.

The possibilities are open, within the constraints imposed by the Wallet model and patterns of use supported by the various parties. Such patterns are built into the interfaces for the services involved. Implementation of these patterns should be factored out from the Service interfaces, so that implementation may be delegated to standard (and extensible) support code.

7 Writing Jtrix applications

The Jtrix base API is deliberately general purpose and minimal. From the above description, it might not be clear how one can write applications using this API. This section introduces some higher level concepts and discusses how they are used to write applications.

First of all we describe a Jtrix application in terms of its responsibilities and the components that it requires (Section 7.1). Next we make a quick survey of the features that Jtrix does provide (Section 7.2), and then go back to look in more detail at those that must be implemented at the application level (Section 7.3).

Finally, we discuss potential mechanisms for factoring out these missing, application-level, facilities so they can be conveniently outsourced to those in a better position to implement them optimally. This is the “Hosting Service” which is discussed in Section 8.

7.1 Taxonomy of an application

By looking at particular applications we can see how they fit into the Jtrix model. We consider two kinds: an end user application is for the benefit of users, while an application providing services exists for the benefit of other applications.

7.1.1 An end user application

An end user application doesn’t provide Jtrix services. Such an application will exist to fulfil some human-oriented function, for example a Web server, shopping portal or game gerver.

Applications are composed of netlets which contain all application code. These netlets may use services provided by other Jtrix applications. They also require hosting services on which to run. An application is responsible for:

- Implementing the end user service.
- Locating hosting services and deploying netlets to them.
- Keeping the state of all netlets consistent.
- Managing communications between netlets.
- Locating other service resources such as storage, DNS names, IP addresses.
- Distributing warrants for these services to the netlets that require them.
- Paying for all services used.
- Creating and destroying netlets to optimise deployment, maintain redundancy, and/or minimise cost as appropriate.
- Managing trust to ensure that risk is controlled appropriately.
- Providing an administrative interface for control and diagnostic purposes.
- Serving the Jar files which implement the application.

7.1.2 Applications providing services

Applications can also implement Jtrix services. Such an application has all the responsibilities of a regular application, plus some additional tasks:

- Negotiating contracts and issuing warrants.

  This may be done on- or off-line. If negotation is done on-line, then it is presumably implemented through some kind of negotiation service. Off-line negotiation is presumably done face to face or through the Web, over the telephone, etc, and is out of the scope of this document.

- Responding to binding requests from clients.

  A netlet which implements the service needs to be selected and a descriptor provided. In may cases, the new netlet will need to communicate with the main body of the application in a secure manner.

7.2 What Jtrix does do

In the face of all that, Jtrix doesn't seem to do much. Here's what it does do:

7.2.1 Inter-application communication

Jtrix standardises the interactions between applications and services with the service model. Services implement a set of service facets. These facets can be classified as follows:

General purpose Appropriate to services of different types. Such facets may be used to implement general "design patterns" for services which may be used to standardise payment models, deployment control information, and other functions which are generally appropriate to any kind of service.

Class specific Appropriate to all services of a specific type. Class specific service facets may be implemented by many different services, allowing the client to substitute one server for another.

Service specific Appropriate to only one implementation of a service. Such facets can be used to provide value added function over and above that provided through a class specific API, or service specific administrative functions.

7.2.2 Proxying and aggregation of services

One service may use another. Warrants and descriptors both contain data which is not used directly by Jtrix and may contain anything, including other warrants and descriptors.

A netlet may pass a facet obtained via one service connection to another netlet via a different service connection. The recipient cannot tell the difference. In this way, services may be directly aggregated. Simple encapsulation is another option.

Warrants can contain the netlet descriptors directly. This provides a simple way to implement a service adapter or aggregator. The warrant may contain one or more other warrants in its opaque data, and the descriptor of the netlet which implements the aggregation. When that netlet is created, it can bind the warrants and implement the aggregated service directly. This technique can be used for any simple service which does not maintain any service state.

7.2.3 Accounting

Service netlets may run in the accounting groups of their clients. This means that they do not need to be concerned with payment or contract information for the resources they use locally, greatly reducing the complexity of the netlet.

Service netlets that do pay for themselves are allowed to provide service to any netlet on the same host.

7.2.4 Binding protocols

The service binding system is the primary security mechanism for services, when combined with node certificates. Nodes and services may cooperate with some certification program for security or reliability, allowing the service to deploy differently depending on the certification status of the node.

The binding protocol allows major deployment decisions at service binding time. Binding may not necessarily involve the downloading of a simple proxy, but the migration of a major part of the application.

7.2.5 Uniform access to services

Whichever node is hosting a netlet, the netlet should have access to any service for which it has a warrant. The service may respond to binding requests with a simple proxy, or a full or partial state download, whichever is appropriate to the situation and security constraints for the service. Different binding protocols may allow varying degrees of interaction between the node and service.

7.3 What Jtrix doesn't do (and why it doesn't)

These key features are missing from the Jtrix API, and need to be implemented at application level.

In general, the end-to-end argument is the justification for leaving them out; they are difficult to solve at the general API level or more efficiently solved at the application level.

In addition, mechanisms implemented as application level tools can be versioned and substituted independently of the base API, with all versions and implementations able to run concurrently.

7.3.1 Intra-application communication

Jtrix manages connections between netlets on the same node, by providing service connections. Communication outside the service framework is not addressed by Jtrix. There are two options for communication between the netlets belonging to an application:

- Basic IP services

  The netlets can implement their own communications, using whatever security protocols are appropriate. Jtrix does allow the basic network facilities required for this.

- Use a communication service

  This functionality can be contracted out. The service chosen to implement the communications should provide the required security.

  Using a service can allow the communications layer to be dynamically changed for different deployments of the same application. It can also allow different applications to share the same communication system.

Inter-netlet communications involve the following processes:

- Keeping track of netlets as they are created and destroyed.
- Discovering and propagating the base layer IP addresses corresponding to each netlet.
- Securing the connections and transmitting the data.
- Detecting the failure of a netlet.

These processes may be substantially different depending on the application. The more dynamic an application, the more difficult it is to keep track of membership.

7.3.2 Redundancy and state management

Applications should maintain their state in a redundant manner. In the event of node failure, the application should be able to recover. State changes should be propagated between redundant copies.

Requirements for consistency and transactional behaviour differ greatly between applications, and methods to implement them are difficult to solve in a general manner.

Applications also need to respond to the failure of a node, and ensure that adequate consistency is maintained.

7.3.3 Contract negotiation

Contracts are a thorny issue. Jtrix requires contracts, but contracts themselves are too complicated to address at the base level. As it turns out, the base level only really requires the warrant abstraction: warrants provide evidence that a contract exists.

Contracts may be negotiated in many ways, and the negotiation may occur inside or outside Jtrix. Jtrix merely requires that the negotiation process result in the issue of one or more warrants.

There is nothing to prevent contract negotiation happening automatically and online, and it is anticipated that this area will receive a lot of attention in the future. Automatic contract negotiation is being worked on by a number of organisations. However, it seems unlikely that it will ever be simple.

It seems better to implement an indirection layer into applications so that manual or automatic systems can be plugged into them as required.

7.3.4 Service binding and code downloading

When a service issues a warrant, it must refer to servers which can respond to the resulting binding requests. The response is generated in an application specific manner. Descriptors refer to code by URL. Both server descriptor servers and codebase servers require a fixed IP address or name which will be valid for the lifetime of the warrants and descriptors issued.

7.3.5 User interface

Jtrix doesn't provide any form of UI.

More mainstream systems have at least a process level interface allowing one to query running processes, list resources in use, and forcibly terminate processes. Jtrix applications require something similar.

Many applications will need an administrative UI specific to the application; this may be implemented using the same mechanisms.

Nodes can offer access to the kind of information and control we require; this access is provided through specialised node facets. Other netlets should be able to gain access to these facets only according to the terms of the hosting contracts involved.

We need to standardise the facets involved for these operations to be universally available. Thus UIs are pluggable and can vary with need.

7.3.6 Security policies

Jtrix allows trust relationships to be established with a node operator before code is downloaded to their nodes. The relationship with the node is the most important trust relationship for an application operator to consider, since all other security is dependent on the node's integrity.

Other than that, Jtrix security operates under a mechanism of transitive trust. Trust is established based on certification by a trusted party; certification must be based on verification.

Jtrix ensures that the transitive trust relationship need not be broken, i.e. it need not contain any holes where one party must assume that another is trusted. If verification is correctly performed, the system should be secure to the extent guaranteed by the verification process.

7.3.7 Run netlets on nodes

The Jtrix base API does not provide any means of executing a netlet on a node other than in response to service binding from that node. It is assumed that higher level hosting services will exist to allow netlets to be deployed from a remote location.

7.3.8 Node Resources

Certain nodes may offer, as part of the contract under which a netlet is running, access to low level resources on that node. These resources are not necessarily Services, because they are not uniformly available across the Jtrix. Location and connection to such resources is not defined in the base layer of Jtrix.

Raw disk space, fixed IP addresses, and hardware accelerators for various protocols or mathematical operations are examples of such resources.

7.3.9 Low level resources

Low level resources are those which are provided as part of a hosting contract. These resources are accessed via the Hosting Service under which a netlet is running, and are billed and accessed under the same contract.

Disk space and networking are the basic low level resources, but any type of resource may in princpal be tied in to a Hosting Service.

These resources are not necessarily services as such, because a warrant is not required by a netlet to access them, and they are not available except to those netlets running under the Hosting Service which controls them.

Applications may be written to require access to any resource as if it were a low level resource. The applications' Hosting Service may subcontract the resource to a "real" Jtrix service if required.

7.3.10 Deployment optimsation

The Jtrix service model enables access to services to be as uniform as possible across the Jtrix. The service binding mechanism allows services to adapt their implementations to the environment to which they are being deployed, in order to allow this uniformity.

This is useful behaviour at the macro level, where large subsystems are being operated by different parties, but consider a single application, which may itself be composed of many subsystems, each of which is an independent Jtrix entity and may be administered as a separate, bought-in service, or as part of a larger application. It is likely that many of these subsystems will have similar deployment and security constraints.

Ideally, we wish to be able to control and optimize the deployment of netlets when considering aggregations of services and applications, in order to maximise performance by identifying and minimising bottlenecks. Thus *must* happen at an application level, and it would be nice for it to happen at a higher level, with optimisation being performed across administrative boundaries.

We'd like a mechanism by which we can locate hosting resources which will allow us to optimise our deployment with respect to the other resources we are using.

7.3.11 Where we go from here

With all these management-level features missing from the core of Jtrix, and yet so fundamental to any application's make-up, it makes sense to define some higher level support. This is discussed next, in section 8.

8 Hosting Services

In this section we introduce an application management API, built on top the base API, called the *Cluster API*.

A Cluster is an association of nodes and node specific resources. First of all, in section 8.1 we discuss the properties of these cluster resources. Next, in section 8.2, an overview is given of the various API's involved. In section 8.3 we look at a design pattern for a simple class of fault tolerant network application.

8.1 Cluster Resources

A Cluster resource is provided by one or more nodes in a cluster. The resource may be available on more than one node, either serially or concurrently. Resources may be used concurrently by multiple netlets.

Resources are classed according to the interfaces they support, and according to resource specific metadata. When searching for a resource, one should inspect the interfaces supported by a resource to discover those resources which are compatible with the application using them. Resource metadata may then be used to discover the suitability of a resource.

For example, an IP Address resource may implement the socket factory API, showing its suitability for use in conjunction with the Nodes trampolining system. Socket factory resources can act as service providers for the platforms native networking API. The metadata for such a resource might include the port numbers available. If a specific port number is required, not all resources implementing the socket factory API may be suitable.

A Resource class may be available on multiple nodes. This can be determined as part of the discovery process. It can also be determined whether the resource is available on more than one node concurrently, or on a set of nodes, and whether that availablility is serial or concurrent.

8.1.1 Resource Instanciation

Before a resource may be used, an instance must be created. An instance is associated with actual system resources. For example, a Network resource may define a port range, but not an actual IP address. There might be hundreds of IP addresses available. Only when the resource instance is created is an IP address actually assigned.

Once the instance is created, the system resources are owned by the contract used to instantiate the resource. The resources will not be released until the resource is released or the contract is terminated.

8.1.2 Resource Connection

Before a resource instance can be used, it must be *connected*. Connection makes an instance available to netlets on a particular node or nodes.

8.1.3 Resource Disconnection

A resource may be forcibly disconnected from a netlet, or from a node. Netlet disconnection clears any session state involved in the use of the resource, and node disconnection may make the resource available on another node.

Returning to our Network resource example, netlet disconnection would close any sockets opened by the netlet, making the TCP or UDP ports available for use by another netlet. Node disconnection would, in addition, possibly allow the IP address to be subsumed by another node.

In the event of node failure, resources are automatically disconnected, and may be available for reconnection on a different node if that is part of the specification of the resource. If the resource is not available on another Node, the resource will be released instead.

8.1.4 Resource Release

Releasing a resource deallocates any system resources used. Resources may not be released while in use.

8.1.5 Using Resources

Once a resource is connected to a particular node, it may be used by netlets subsequently created on that node. There are two mechanisms by which a resource may be used by a Netlet:

- The resource may be mapped to a symbolic name when a netlet is executed. Such resources are available by name.
- The resource may be discovered dynamically.

8.2 Cluster API's

The Cluster API covers four separate areas, each of which might be used independently or substituted for alternative mechanisms

8.2.1 Node Management API

A cluster implementation may control access to node level resources.

Such resources might include disk space, fixed network addresses, and hardware accelerators. This part of the API is implemented by node management software to export such resources to a cluster control system.

8.2.2 Cluster Control API

This part of the API allows consumers to utilise a cluster and its resources.

The cluster operator issues a Warrant for this API which is bound by consumers. The resulting control interface may be used to allocate resources, and run netlets, and monitor status. Users of this interface are called *Application Controllers*, and netlets run using this interface are called *Managed Netlets*. Application Controllers are not necessarily Managed Netlets.

Measures may be taken to ensure that an application controller always exists. The cluster may be provided with a Warrant through which it can create an application controller if necessary. This mechanism is lease based, so that if an existing controller fails to renew its lease, a new controller will be created by binding the warrant. This mechanism detects both explicit failures, where the controller netlet terminates, or subtle failures, where the controller netlet merely malfunctions.

The API supports aggregation, allowing virtual clusters to be created using lower level clusters, in a manner transparent to the users of the cluster, and may not be implemented by specialised Hosting Services, which may substitute it for extended definitions of Managed Netlets and their behaviours.

8.2.3 Resource Discovery API

Managed netlets have access to a resource discovery API through which they may use symbolic names to access either cluster resources or regular Jtrix Services. The mapping from symbolic name to resource or service is defined through the control API when the netlet is run. This API may be extended in the future to allow dynamic discovery of resources.

It is up to the application logic to obey the constraints imposed by the use of such resources. If an application netlet uses a persistent resource, then it is necessary for that resource to be available on any node on which the netlet is to be run. Services should in turn be prepared for a resource to be unavailable.

Such resources remain the property of the hosting contract. The controller may release such resources at any time, terminating access to the resource by managed netlets.

8.2.4 Generalized status API

The API allows for system derived status information about netlets to be retrieved, in much the same way that traditional operating systems allow various aspects of a process to be inspected. Such information includes resource usage counters. Status information is provided as an SNMP-like MIB, allowing complete extensibility. Running netlets may export arbitrary typed status information which is made available to the application controllers as part of the MIB.

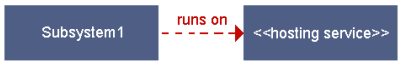


Figure 3: A subsystem running on a hosting service. The subsystem may comprise many netlets. The agency which started the subsystem is not shown.

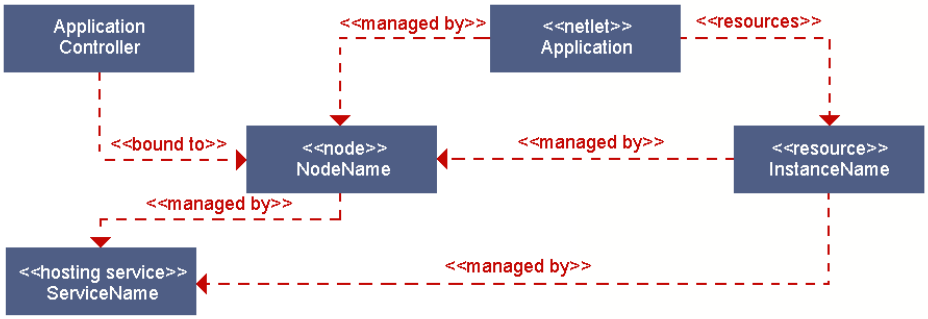


Figure 4: A netlet running on a hosting service, under the management of an application controller. The netlet is running on a particular node, and using a resource supplied by that node under the management of the hosting service.

8.3 Typical Scenario

The Hosting API is designed to be low level enough that middleware systems can be built on top of it, and high level enough to be useful for applications without supporting middleware.

Figures 3 through 8 show the various ways we depict relationships between hosting services, netlets, and resources.

As an illustration, consider a simple pattern for Jtrix services. Applications following this pattern have the following characteristics:

- The use a Helper service which acts as application controller

  The Helper service deals with boilerplate details like service advertisement, user interface, codebase download, and master election. It may be an entirely separate service, or be created to run only this application.

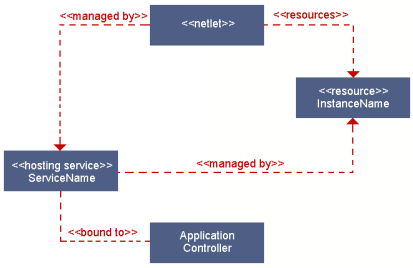


Figure 5: The same situation as in figure 4, except that the node isn't important and isn't shown.

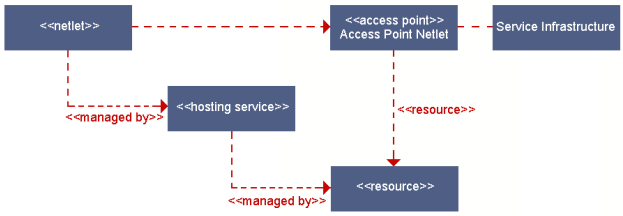


Figure 6: A Netlet using a service. The access point is downloaded and runs under the same hosting service as the consumer netlet. The service is using a resource from the hosting service. It has been passed that resource by the consumer netlet, through the service interface.

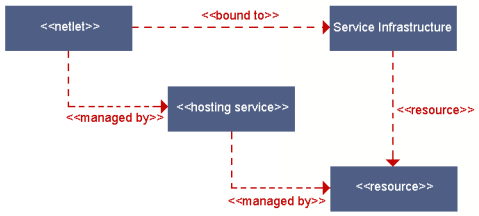


Figure 7: The same situation as in figure 6, in an abbreviated notation.

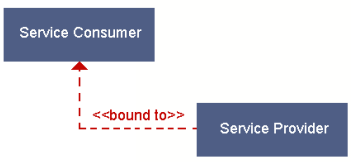


Figure 8: A service being provided from provider to consumer. There may be multiple service connections and access points involved in the connection.

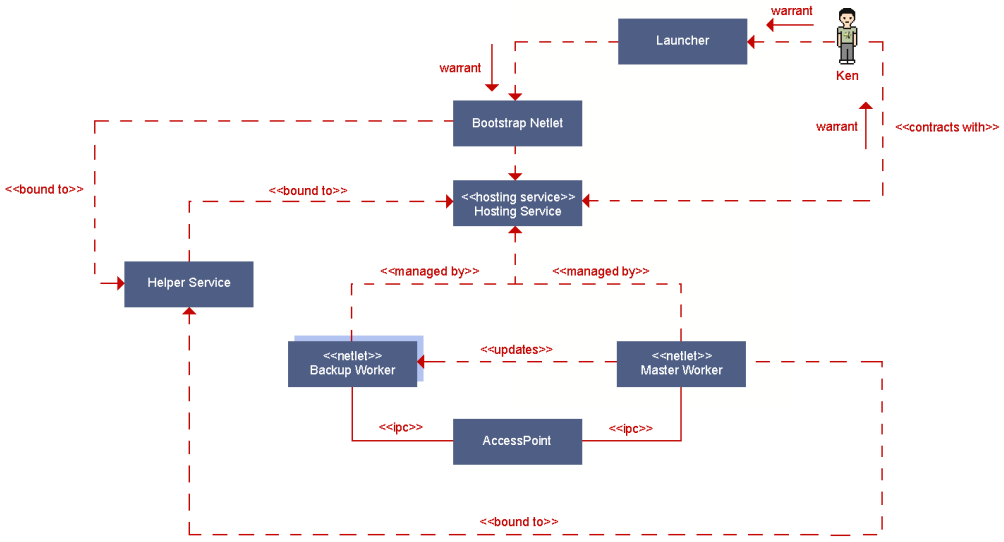


Figure 9: A typical Jtrix service style application

- They maintain a group of worker netlets each of which maintains a copy of the application state.

  There may be different types of worker. Each worker type may be associated with resource instances. When a worker fails, it should be replaced with a worker connected to the same resources, if possible.

- Exactly one worker is designated as a master.

  Only workers of a particular type are eligibile to be master. The other workers of that type are designated as backup masters

  All changes to application state occur on the master and are propagated from the master to the backups. In the event of master failure, a new master is elected from the available backups. The most up to date backup is preferred.

- If the application provides an external service, the service may optionally be supplied by any worker, or by the master alone.

  Some operations *must* be performed by the master. These operations require access to up to date state information, for state integrity purposes. These operations are routed from backup controllers to the master over worker-to-worker RPC. Other operations may be performed using the (possibly out of date) backup copy of the state available on any worker. These decisions are application specific.

- If the application provides a Jtrix service, the service is provided to clients via access point netlets.

  Access point netlets use RPC to communicate with a worker. The criteria for selecting the worker are as above.

We denote the person responsible for maintaining the application as the "Application Administrator," or "KEN" for short.

The lifecycle of such an application will involve the following steps:

1. Contract Negotiation

KEN must first obtain a Warrant for a hosting service. This involves some kind of contract negotiation, which may be explicit or implicit, inside or outside of Jtrix. The Warrant itself is merely an XML document. Some possibilities are:

- KEN runs his own cluster. He can then issue himself a warrant.
- KEN obtains a warrant directly from a Hosting Service Provider.
  He uses a web interface to agree to the contract, make payment provision, and download the warrant. Payment may have to be made when the service is used, via a Wallet service, or the wallet may have to be provided to the web site. Maybe just a credit card is required.
- KEN obtains the warrant from a broker.

- KEN is working for some organisation which runs a cluster. He gets a warrant as part of his work.

1. Application Deployment

   KEN now initialises his application. This involves the creation of a Bootstrap Descriptor which describes the application's bootstrap netlet, which is a special netlet written to start the application. Applications will be distributed with a mechanism to create this initial descriptor. This mechanism may require arguments to configure the application, or may be a complicated GUI style app. The bootstrap netlet will need, as a minimum, the warrant for the hosting service.

   KEN uses a launcher application to start the application. The launcher application starts the bootstrap netlet, using the Bootstrap Descriptor. A complicated application may include its own launcher as part of the installation mechanism, bypassing the use of an explicit Bootstrap Descriptor.

   The launcher application stays around long enough to ensure that the bootstrap has completed successfully, or to report the reason for failure. It can do this by interacting directly with the bootstrap netlet.

   The bootstrap netlet uses the hosting warrant to create the first worker netlet. It can use the hosting services status reporting mechanism to retrieve initialisation dependent objects from the worker netlet, or the reason for startup failure.

   In order to bind the hosting service warrant, the launcher application needs to provide a node environment. A convenient way of doing this is to embed nodality in the laucher.

2. Bootstrap

   Once the bootstrap netlet is started, it must initialise the application, and provide any required feedback to KEN.

   (a) Helper service initialisation
   Helper services used by the application may need to be initialised. If a helper service is used to serve the application codebase, then the bootstrap should upload the codebase at this point. The filenames for the codebase and the warrant for the helper service (and any other configuration parameters necessary to the application) are supplied during the creation of the bootstrap descriptor. The interaction mechanism used to control the bootstrap process may be used to read or write files during the bootstrap process.

   A helper service may also be used to handle binding requests, and that service may also be initialised at this point. It may be necessary to initialise hosting resources or helper services in order to generate an initial warrant for the applications own service, for instance, to find binding protocol addresses if these are not known in advance.

   If the helper service is to run on the same hosting service, then it should be bootstrapped separately, before the application is bootstrapped.

3. Worker initialisation

   Application initialisation consists of using the hosting service to run the first application worker, which will become the master worker. Resources may need to be instantiated, connected and mapped for the controller use. Helper services may also be mapped as resources, if required.

1. Feedback to KEN

   Once the worker has initialised, any required feedback may be picked out of the controllers status via the hosting service, and reported back to KEN. Similarly, any failure feedback can be reported back.

   Once the controller has sucessfully started, the bootstrap netlet can exit,and the launcher program can terminate.

1. First Controller startup

   (a) Service advertisement

   If the application provides a Jtrix service, it must advertise that service. This may be done internally, or more likely, via a helper service. The helper service will likely need configuring to properly service the bind requests for this application.

   (b) Backup integrity

   The first controller should arrange for at least one backup if the application is to be fault tolerant. RPC links must be established to backups, and initial state propagated if necessary.

   The hosting services also allow for an auto bootstrap mechanism if a control lease expires. This lease may have only one active lessor, and this mechanism may be used in a master election process if necessary. The autobootstrap mechanism may be used to reconnect "orphaned" parts of an application by binding an external service. If this mechanism is to be used, it should be initialised, and the lease periodically renewed.

   (c) External services

   If the application provides an external service, then the listener should also be established at this point.

   (d) Feedback

   Any necessary information about the deployment of the service may be fed back to KEN via the Hosting services status reporting mechanisms.

2. Run time events

   (a) Backup controller netlet or node failure

   If a backup controller fails, due to netlet or node malfunction, it should be replaced. Ideally, it should be replaced with a new netlet bound to the same resources. If this is not possible, the state must be reinitialised from another backup or the master. This is shown in figure 10.

   Reuse of the resource might be worthwhile for a number of reasons, including the preservation of an IP address, or to avoid copying a large amount of data.

3. Master controller netlet or node failure

   If the master node fails, a new master should be elected from the remaining backups. The new master controller should follow a startup sequence similar to that detailed for the first master controller.

1. Service Binding

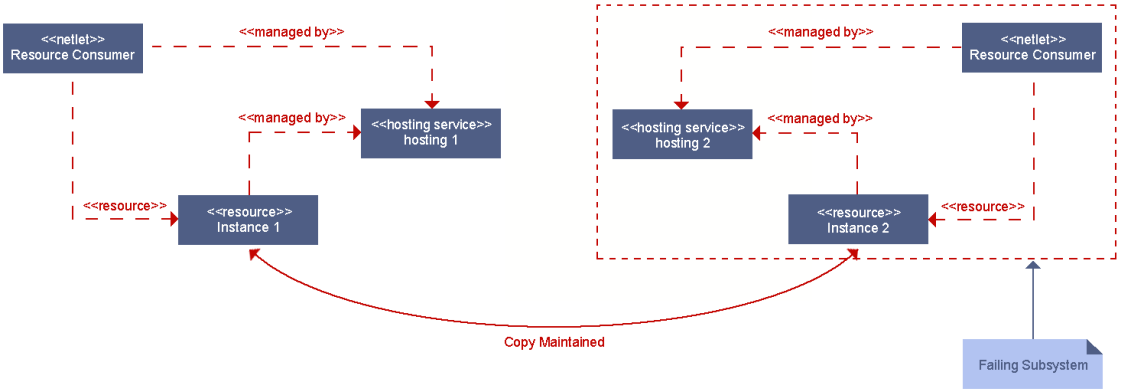
   Service binding will result in the creation of access point netlets. These netlets will make network requests which must be serviced.

2. External Services

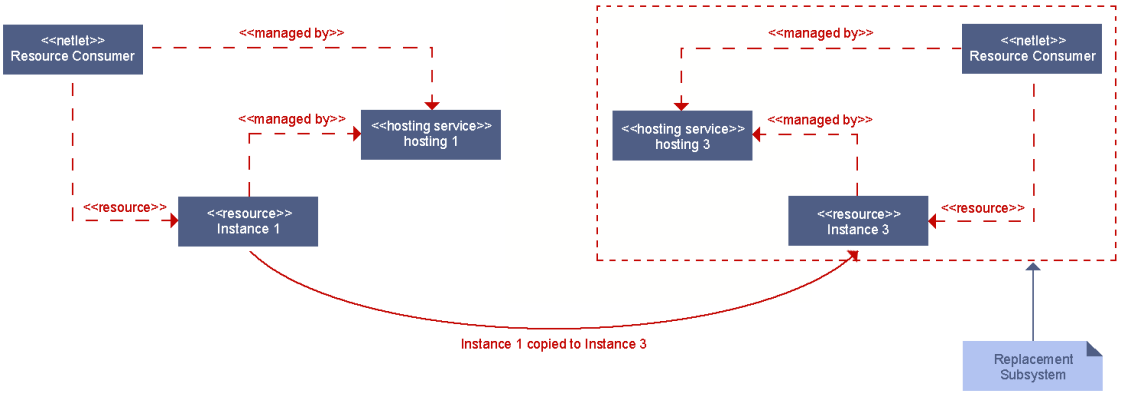
   The master and backups may have to service external network requests.

1. Application Shutdown

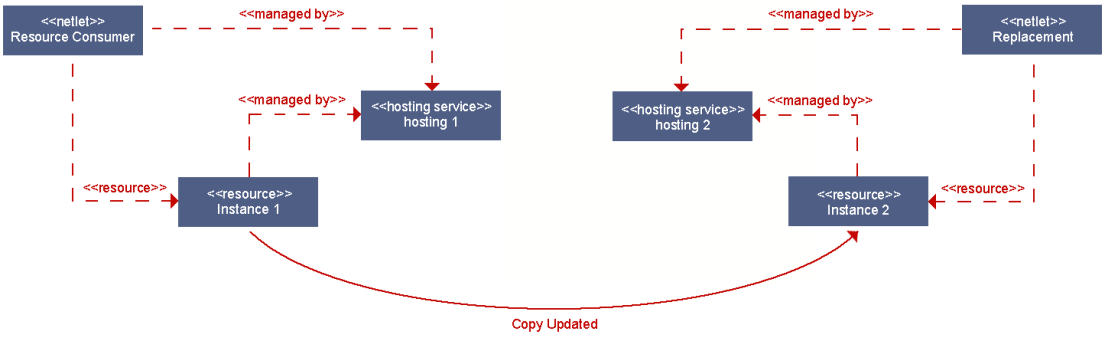
   At some point, the application will need to be terminated. The master should arrange for all netlets to terminate.



(a) Starting point. Two copies of the data exist, in two different resources.



(b) Basic failover. The resource contents have to be recreated in a different resource.



(c) Optimal situation. A new netlet can be created with the original resource.

Figure 10: Reusing an exiting resource if possible.

The hosting warrant can be used to reconnect to the hosting service if desired, for manual status inspection or forced termination. Standard user interfaces can be used for this, much as a traditional operating system provides standard interfaces for process inspection or termination.

This pattern is used by many of the basic applications and services which form the Jtrix core software suite. It seems suitable for fault tolerant, client/server based network applications. Jtrix provides a support framework (see section 11.3.4) which handles much of the boilerplate involved in such an application.

Other patterns and corresponding frameworks are of course possible.

8.4 Virtual Hosting Services

The Hosting API makes it possible to define a "Virtual" Hosting service. Such a service aggregates the services provided by one or more other Hosting Services. This has the following advantages for an application:

- The application can be migrated between real hosting services, or spread across services from one or more providers.
- The creator of the virtual hosting service can issue multiple hosting contracts for the virtual service, keeping different components of an application separate. Only one contract is required from the provider of a real Hosting Service.

The Hosting API's are designed to make this aggregation possible.

9 Locality Dependent Services

In this section, we look at services which have some kind of constraint with respect to the location of their consumers.

9.1 Example Scenario

As an example, consider an HTTP listener service which fulfills the requirements set in the Harrys Game example in section 4. Such a service would be able to optimise the path between Harry's web servers and the end user, in two ways:

- Choosing the most optimum server for the particular user
- Creating new servers in response to changing usage trends

If the listener encapsulates some kind of HTTP accelerator or load balancer, typically the requests may be routed only to nodes within a particular network. The IP level NAT involved in the load balancing imposes this constraint. Also, the service may be more easily implemented if all networking is performed in the domain of the service provider.

Of course, consumers of such a service might download service proxies, from the optimised endpoints, but that seems a little pointless. One goal of such a service, after all, is to provide a level of abstraction which allows the service to provide smart optimsation of the service, and introducing the extra network hop is counter-productive. To do this, it needs to place constraints on where the consumers shall run.

9.2 A Pattern

We propose a pattern for the design of services such as this. The pattern is as follows:

- We call the location constrained service the "Primary Service"

  The primary service is presented as a hosting service, with a specialised control API. The standard control interface is not available. Instead, a customized interface is made available.

- The operations performed by the service consumer are encapsulated in a different service interface.

  This service interface is called the "Back End Interface." The consumer of the primary service must itself supply a service which implements this interface. This service is called the "Back End Service."

- The Primary service determines where its consumers are to be run, in its hosting service.

  The primary service offers a specialised Control Interface, allowing the consumer to define the back end service, and the local resources it requires to run. The definitions include the constraints under which the back end service will be created. In general, constraints will be loose enough to allow the service to optimse deployment.

- The primary service service is contracted normally, i.e, a Warrant is provided to the consumer by some mechanism.

  This warrant allows access to the control elements of the service, and is the warrant for the "Primary Contract"

- All Hosting charges are reflected according to the Primary Contract.

9.3 Scenario Revisited

We propose an HTTP listener service conforming to the pattern above. The Primary service interface allows the consumer to supply base URL's, and supply backend services to handle requests prefixed with the given base. Legal URL's are determined by the HTTP service. The service may work in association with a consumer supplied DNS service to enable arbitrary URL's to be served.

The HTTP service may offer arbitrary resources to its consumer netlets, in addition to the HTTP functionality.

It is possible to implement this service using generic techniques. It is also possible to encapsulate external HTTP optimisers.

10 Harry's trading game revisited

Let's take a look at this hoary old favorite from our new perspective.

10.1 Basic Architecture

We'll look at the basic architecture for the web server part of Harry's game. An HTTP service as described above is used. It relies on a DNS server, provided by Harry, to maintain redundant sites, substituting new IP addresses for the site name as required. It also tries to use redundant IP address resources to minimise the need to do this.

Harry uses a web server to serve the content. The web server acts as the back end for the HTTP service, and is configured with war files which contain the content and harry's servlets. The Web server consists of a control system, which runs outside the HTTP service in its own Hosting

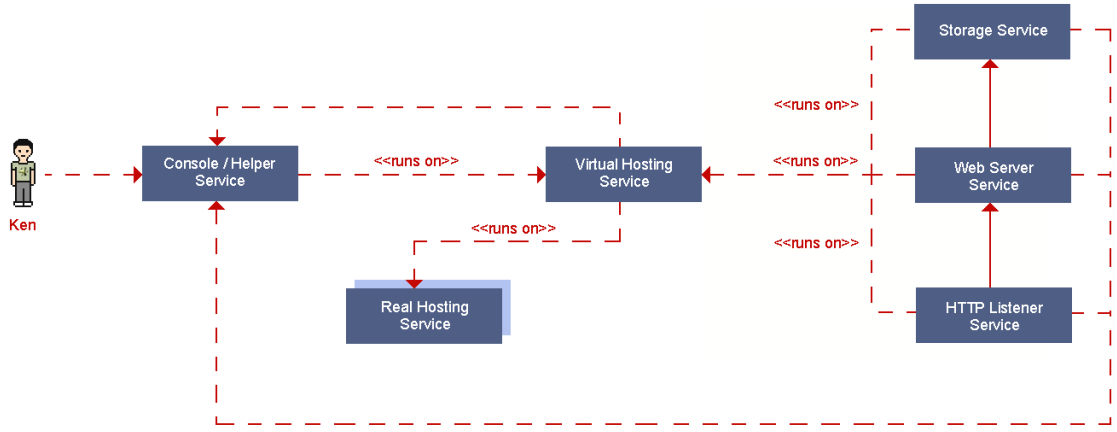


Figure 11: Functional Overview of Harry's Game

contract, and back end netlets, which are created by the HTTP service and contain the actual web server.

Both web control system and web server back end netlets use a storage service. This also consists of a control system, running separately under its own hosting contract, and access point netlets, which provide the storage service to clients. The storage control system requires file space to store the actual data. This is obtained as a resource from the storage Hosting Contract.

The storage access points can use file system storage to cache data. This may be obtained through service connection between the Web Server and the storage system. The Web server may obtain the resource from the HTTP service through the Resource Discovery mechanism.

The Service and Resource relationships in Harry's game are depicted in figure 11.

10.2 Initial Deployment

Initially, Harry uses a bought in DNS Service and generic components for all the other components. Eventually, he will buy in a Hosting Service. He obtains warrant for all the services from the web sites run by the providers.

The application is developed on harry's own machine, by using the Jtrix implementation of these services. Harry uses the commercial DNS service from the very beginning, so he doesn't have to change URL's. Bootstrapping the services is an interesting process, because the virtual hosting service uses the console service, which in turn runs on the hosting service. It is possible because the hosting service can be made to connect to helper services after startup.

After development, Harry can migrate the application to the Internet merely by adding the new Hosting service as a supplier for his virtual service, and removing his development system from that service.

The virtual console service is migrated along with the rest. When it moves, Harry can find its new address by looking at the status information from the hosting service.

The initial deployment is shown in figure 12.

This deployment does not optimise.

10.3 Optimising Deployment

Eventually, Harry want to improve his web site performance, so he contracts a commercial HTTP service, which offers hardware based load balancing. Migration is simple, merely passing the DNS

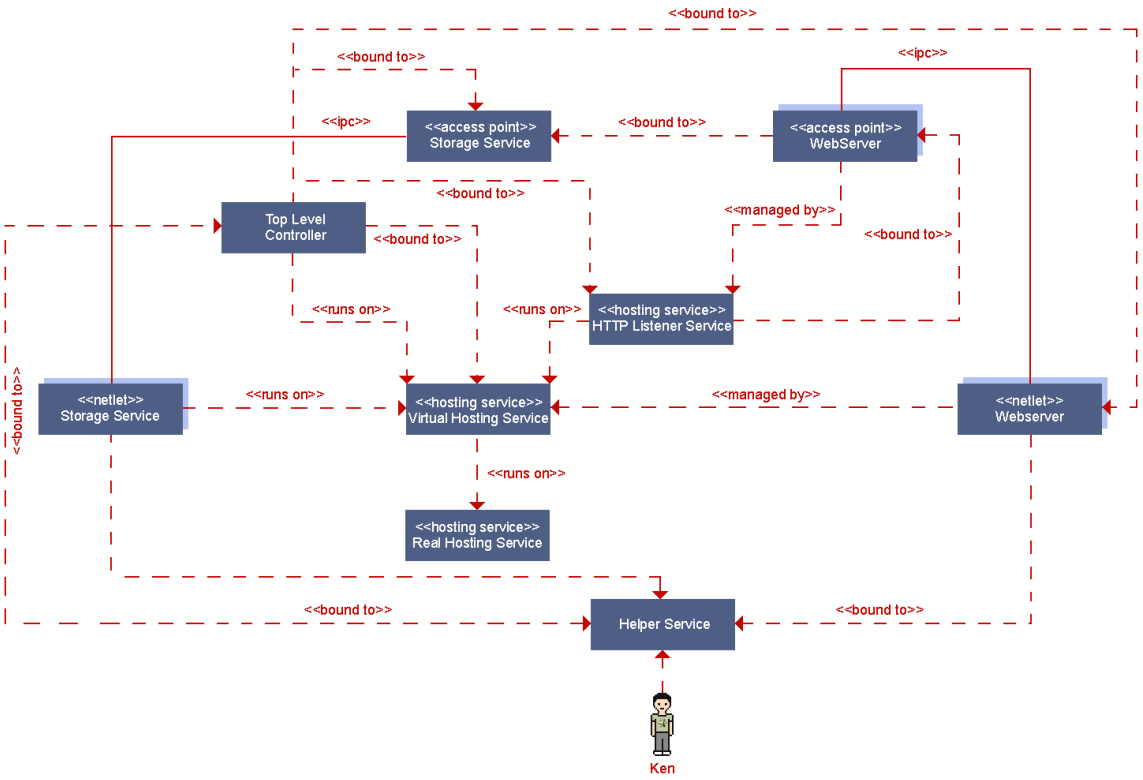


Figure 12: Initial deployment of Harry's Game

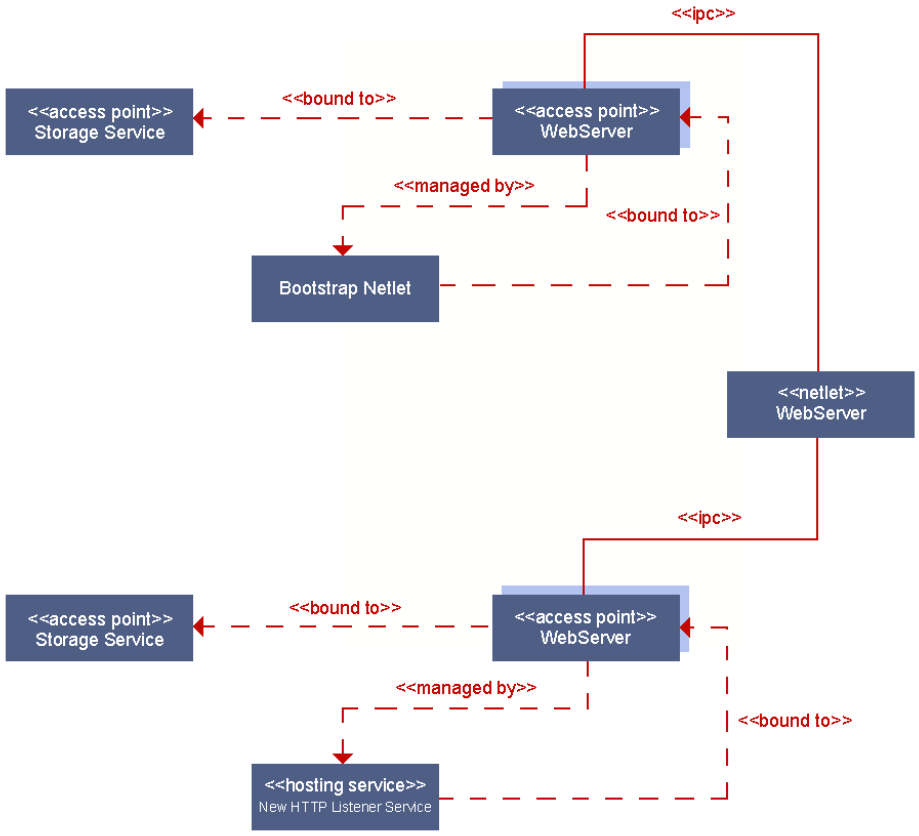


Figure 13: Migration to a new HTTP service. The new service can run in parallel to the old during the migration process.

warrant and defining the URL's for the back ends. When the service is running, Harry cancels the URL's in the old HTTP service and closes it down. This migration is shown in figure 13.

This deployment may optimise. The HTTP service is free to create new backends as required.

11 Reference implementations

What Jtrix.org is working on, and the purposes for which the projects may be used.

11.1 Jtrix

The base API is available under an open source licence. It defines basic, vendor-independent, interfaces and services upon which specific nodes and services may be built. The community will control the development direction and encourage the development of standard service interfaces.

11.2 Nodality

Nodality is an implementation of a Jtrix node. It is available under an open source licence and offers the following features:

- Linux and Windows platforms

  Currently, Nodality runs under Linux and on Windows. Under Linux, the basic networking support provides ARP spoofing to allow highly available clusters.

- Accounting Support

  Supports resource accounting so that users can be charged for runtime resources such as CPU and memory.

- Extensible through administrative netlets

  Can be extended by executing special administrative netlets. Nodality delegates many functions to these netlets. These administrative netlets implement the hosting contracts, charging models, network protocols, and specialised facets of the netlet/node services.

- Embeddable in other applications

  Nodality can be embedded in other applications, allowing them to access Jtrix resources.

  Nodality could be embedded in a Web server, for instance, allowing portal style Web sites to import Jtrix services to enhance existing functionality. Or it could be embedded in a command line or GUI application. Nodality provides an API to the embedding application which allows that application to behave as a netlet.

  The external application can provide specialised functionality or administrative resources to the node if desired.

- Standalone for commercial hosting

  Nodality is also packaged as a standalone server application for use in backend nodes. The management framework allows many nodes to be administered together.

- GUI for debugging support

  Nodality has also been packaged into a GUI which allows multiple nodes to be run in the same application. It displays the status of the netlets running in each node, shows service connections, and allows access to the debugging traces. Useful for demos and testing, as well as being an example of how Nodality may be embedded into another application.

- Low level service support

  Disk resource support is optional. Networking is supported. These services are offered to clients through the management framework, and need not be uniformly available through the cluster.

11.3 Base Infrastructure

The base infrastructure suite consists of the following components:

11.3.1 Jtrixd

Jtrixd is a node implementation, based on Nodality, which is specially designed for large scale cluster operations. Jtrixd implements the node management interfaces of the Cluster API to allow the node to be administrated as part of a large group, and makes basic system resources such as disk space and IP addresses available to clients, through the cluster.

11.3.2 Cluster Controller

The Cluster Controller provides a hosting service, allowing node administrators to maintain a fault tolerant group of nodes and provide contract based access to third parties. The Cluster Controller manipulates member nodes through the Node administration section of the API. Each Node runs a

special netlet which has access to Nodality's administrative API and to the Node Management API. The Netlet uses these to control the node.

The Cluster Controller supports an administrative interface on top of the basic Hosting API which may be used to administrate the cluster as a whole. Administrative functions include contract management, node shutdown, and resource allocation.

11.3.3 Command line API

The infrastructure suite also includes a command line API which can be used to manipulate a hosting service for which a warrant is held.

11.3.4 Support Framework

The Support Framwork is a framework to simplify development of Jtrix applications. It provides tools for starting, controlling, and debugging applications. The support framework consists of a collection of support code to be used in applications, Infrastructure services which perform some of the work but need not be programmed by the user, and management protocols to facilitate communication between parts of an application and with the infrastructure. The support framework also includes some standalone tools such as an application launcher.

11.3.5 Service Advertisement Service (SAS)

SAS is a service which provides support functions for other services. It provides a reliable, redundant server to answer service binding requests, and codebase download requests.

Applications upload codebase files into SAS. SAS provides a corresponding URL which can be used in Netlet descriptors to download the file.

Service binding requests may be handled either using static data registered with SAS, or, for more dynamic situations, a listener may be registered to provide the binding data.

11.3.6 DNS service

The DNS service provides a reliable, warrant based DNS server. Clients may be issued a Warrant allowing them to control a particular sub domain of the top level domains controlled by the server.

11.3.7 Console Service

A Service which may be used by applications to present an administrative user interface. Many applications may be maintained by one console service. The console service may also act as an application launcher.

11.4 Storix

Storix is Jtrix.org's reference implementation of a storage API. It uses a two tier client/server model, with data being stored on redundant servers and accessed through caching or non-caching client proxies.

Storix offers warrant based access control to allow operators to give restricted access to defined parts of the data hierarchy.

Storix supports checkpointing, which can be used in turn by applications which need to checkpoint.

Storix both uses and implements the low level Jtrix storage API, and is written to the Hosting APIs. It can be run under any Hosting Service, and can be optimised by Hosting Services which support optimisation of low level storage.

It implements this API so that it can be used via the storage trampoline mechanism by application code which is written to use the normal language mechanisms for disk I/O.

11.5 Spondulix

Spondulix is Jtrix.org's reference wallet implementation.[6]

Spondulix operates a simple account based financial model, with a two level security system which ensures that account details are kept only in trusted places, but allows financial transactions to take place anywhere in the Jtrix. A single deployment of Spondulix can support many accounts.

Each Spondulix service supports a single currency. Spondulix is suitable for applications where a single virtual currency is required, such as closed economies or application testing.

11.6 Webtrix

Webtrix is Jtrix.org's reference web server implementation.[7] It is available under an open source licence and offers the following features:

- JSP/servlet execution.
- Virtual hosting by name and IP address.
- Servlets have access to Jtrix services.
- Redundant and self-scaling.
- Constructed of general purpose re-usable components .

  Any of these components can be replaced to enhance functionality or used inside other Jtrix applications.
- Can itself be used as a Jtrix service to offer warrant based Web access to Jtrix applications.

[6] Name subject to change without notice.
[7] Name subject to change without notice.