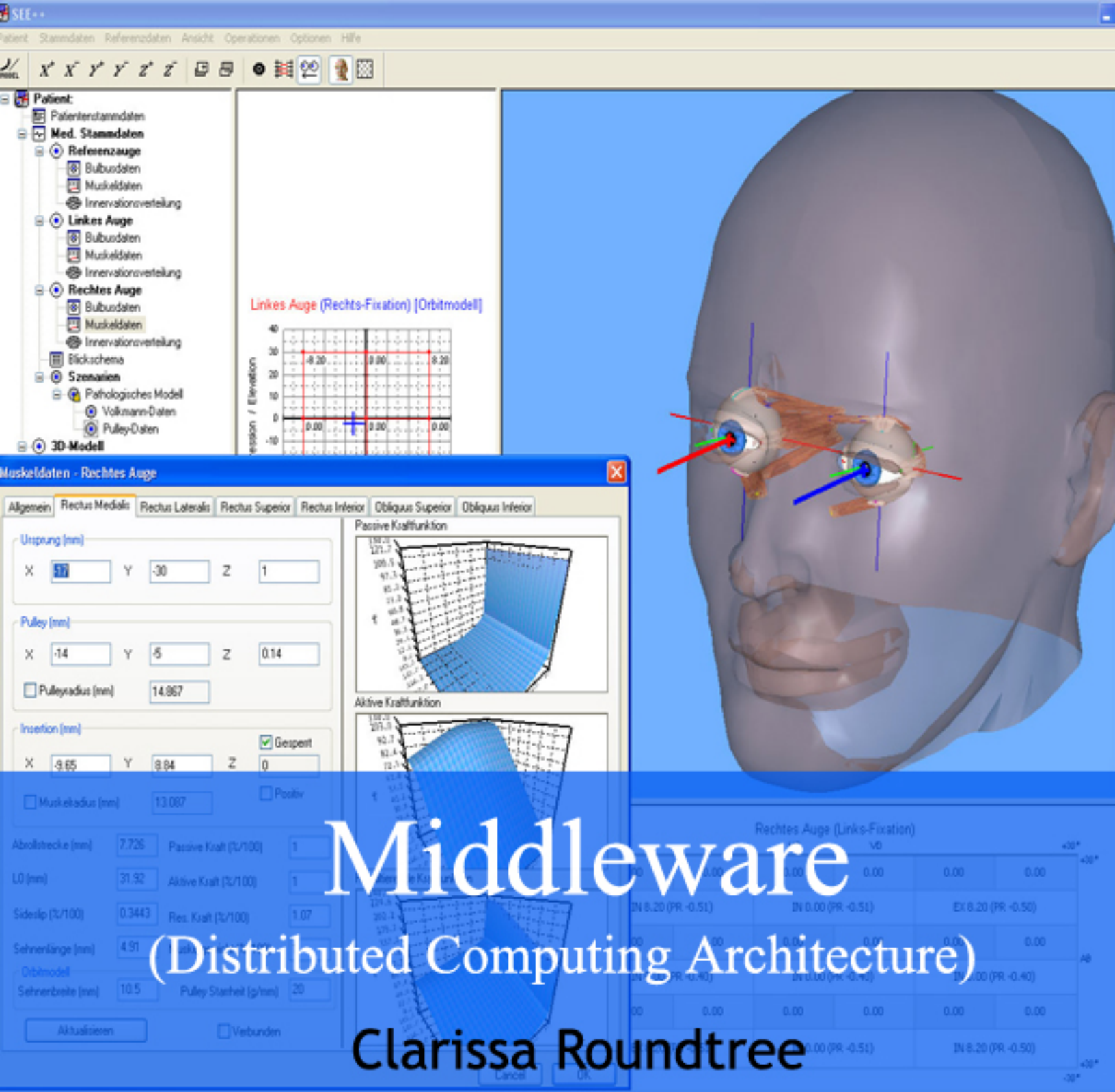

Middleware
(Distributed Computing Architecture)
Clarissa Roundtree

First Edition, 2012

ISBN 978-81-323-3494-1

© All rights reserved.

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1 - Middleware

Chapter 2 - gLite and Flow (Software)

Chapter 3 - Advanced Message Queuing Protocol

Chapter 4 - IBM WebSphere Message Broker and HP RTR

Chapter 5 - Internet Communications Engine and Volunteer Computing

Chapter 6 - Message-oriented middleware and SynfiniWay

Chapter 7 - Yaim and Remote Procedure Call

Chapter 8 - OpenLink ODBC Drivers and Oracle Fusion Middleware

Chapter 9 - ProActive

Chapter 10 - Jsonwsp

Chapter 11 - SOAP and SOAPjr

Chapter 12 - JSON-RPC and Java Remote Method Invocation

Chapter 13 - XML-RPC and RPyC

Chapter-1

Middleware

Middleware is computer software that connects software components or some people and their applications. The software consists of a set of services that allows multiple processes running on one or more machines to interact. This technology evolved to provide for interoperability in support of the move to coherent distributed architectures, which are most often used to support and simplify complex distributed applications. It includes web servers, application servers, and similar tools that support application development and delivery. Middleware is especially integral to modern information technology based on XML, SOAP, Web services, and service-oriented architecture.

Middleware sits "in the middle" between application software that may be working on different operating systems. It is similar to the middle layer of a three-tier single system architecture, except that it is stretched across multiple systems or applications. Examples include EAI software, telecommunications software, transaction monitors, and messaging-and-queueing software.

The distinction between operating system and middleware functionality is, to some extent, arbitrary. While core kernel functionality can only be provided by the operating system itself, some functionality previously provided by separately sold middleware is now integrated in operating systems. A typical example is the TCP/IP stack for telecommunications, nowadays included in virtually every operating system.

In simulation technology, *middleware* is generally used in the context of the high level architecture (HLA) that applies to many distributed simulations. It is a layer of software that lies between the application code and the run-time infrastructure. Middleware generally consists of a library of functions, and enables a number of applications– simulations or federates in HLA terminology—to page these functions from the common library rather than re-create them for each application.

Definitions

Software that provides a link between separate software applications. Middleware is sometimes called plumbing because it connects two applications and passes data between

them. Middleware allows data contained in one database to be accessed through another. This definition would fit enterprise application integration and data integration software.

ObjectWeb defines middleware as: "The software layer that lies between the operating system and applications on each side of a distributed computing system in a network."

Origins

Middleware is a relatively new addition to the computing landscape. It gained popularity in the 1980s as a **solution to the problem of how to link newer applications to older legacy systems,** although the term had been in use since 1968. It also facilitated distributed processing, the connection of multiple applications to create a larger application, usually over a network.

Organizations

IBM, Red Hat, and Oracle Corporation are major vendors providing middleware software. Vendors such as Axway, SAP, TIBCO, Informatica, Pervasive and webMethods were specifically founded to provide Web-oriented middleware tools. Groups such as the Apache Software Foundation, OpenSAF and the ObjectWeb Consortium (now OW2) encourage the development of open source middleware. Microsoft .NET “Framework” architecture is essentially “Middleware” with typical middleware functions distributed between the various products, with most inter-computer interaction by industry standards, open APIs or RAND software licence. Solace Systems provides middleware in purpose-built hardware for implementations that may experience scale or speed limitations when using software.

Use of middleware

Middleware services provide a more functional set of application programming interfaces to allow an application to:

- Locate transparently across the network, thus providing interaction with another service or application
- Filter data to make them friendly usable or public via anonymization process for privacy protection (for example)
- Be independent from network services
- Be reliable and always available
- Add complementary attributes like semantics

when compared to the operating system and network services.

Middleware offers some unique technological advantages for business and industry. For example, traditional database systems are usually deployed in closed environments where users access the system only via a restricted network or intranet (e.g., an enterprise’s internal network). With the phenomenal growth of the World Wide Web, users can

access virtually any database for which they have proper access rights from anywhere in the world. Middleware addresses the problem of varying levels of interoperability among different database structures. Middleware facilitates transparent access to legacy database management systems (DBMSs) or applications via a web server without regard to database-specific characteristics.

Businesses frequently use middleware applications to link information from departmental databases, such as payroll, sales, and accounting, or databases housed in multiple geographic locations. In the highly competitive healthcare community, laboratories make extensive use of middleware applications for data mining, laboratory information system (LIS) backup, and to combine systems during hospital mergers. Middleware helps bridge the gap between separate LISs in a newly formed healthcare network following a hospital buyout.

Wireless networking developers can use middleware to meet the challenges associated with wireless sensor network (WSN), or WSN technologies. Implementing a middleware application allows WSN developers to integrate operating systems and hardware with the wide variety of various applications that are currently available.

Middleware can help software developers avoid having to write application programming interfaces (API) for every control program, by serving as an independent programming interface for their applications. For Future Internet network operation through traffic monitoring in multi-domain scenarios, using mediator tools (middleware) is a powerful help since they allow operators, searchers and service providers to supervise Quality of service and analyse eventual failures in telecommunication services.

Finally, e-commerce uses middleware to assist in handling rapid and secure transactions over many different types of computer environments. In short, middleware has become a critical element across a broad range of industries, thanks to its ability to bring together resources across dissimilar networks or computing platforms.

In 2004 members of the European Broadcasting Union (EBU) carried out a study of Middleware with respect to system integration in broadcast environments. This involved system design engineering experts from 10 major European broadcasters working over a 12 month period to understand the effect of predominantly software based products to media production and broadcasting system design techniques. The resulting reports Tech 3300 and Tech 3300s were published and are freely available from the EBU web site.

Types of middleware

Message-oriented Middleware

Message-oriented middleware is middleware where transactions or event notifications are delivered between disparate systems or components by way of messages, often via an enterprise messaging system.

Enterprise messaging system

An enterprise messaging system is a type of middleware that facilitates message passing between disparate systems or components in standard formats, often using XML, SOAP or web services.

Message broker

Part of an enterprise messaging system, message broker software may queue, duplicate, translate and deliver messages to disparate systems or components in a messaging system.

Enterprise Service Bus

Enterprise Service Bus (ESB) is defined by the Burton Group as "some type of integration middleware product that supports both MOM and Web services".

Content-Centric Middleware

Content-centric middleware provides a simple provide/consume abstraction through which applications can issue requests for uniquely identified content, without worrying about where or how it is obtained. Juno is one example, which allows applications to generate content requests associated with high-level delivery requirements. The middleware then adapts the underlying delivery to access the content from the source(s) that are best suited to matching the requirements. This is therefore similar to Publish/subscribe middleware, as well as the Content-centric networking paradigm.

Hurwitz classification system

Judith Hurwitz created a classification system for middleware in her article *Sorting Out Middleware*.

Remote Procedure Call

With Remote Procedure Call middleware, a client makes calls to procedures running on remote systems. Can be asynchronous or synchronous.

Message Oriented Middleware

With Message Oriented Middleware, messages sent to the client are collected and stored until they are acted upon, while the client continues with other processing.

Object Request Broker

With Object Request Broker middleware, it is possible for applications to send objects and request services in an object-oriented system..

SQL-oriented Data Access

SQL-oriented Data Access is middleware between applications and database servers.

Embedded middleware

Embedded middleware provides communication services and integration interface software/firmware that operates between embedded applications and the real time op.

Other

Other sources include these additional classifications:

- Transaction processing monitors — Provides tools and an environment to develop and deploy distributed applications.
- Application servers — software installed on a computer to facilitate the serving (running) of other applications.

Chapter-2

gLite and Flow (Software)

gLite

gLite is the middleware stack for grid computing used by the CERN LHC experiments and a very large variety of scientific domains. Born from the collaborative efforts of more than 80 people in 12 different academic and industrial research centers as part of the EGEE Project, gLite provides a complete set of services for building a production grid infrastructure. gLite provides a framework for building grid applications tapping into the power of distributed computing and storage resources across the Internet. The gLite services are currently adopted by more than 250 Computing Centres and used by more than 15000 researchers in Europe and around the world (Taiwan, Latin America etc.).

History

After prototyping phases in 2004 and 2005, convergence with the LCG-2 distribution was reached in May 2006 when gLite 3.0 was released and became the official middleware of the EGEE project.

Middleware description

Security

The gLite user community is grouped into Virtual Organisations (VOs). A user must join a VO supported by the infrastructure running gLite to be authenticated and authorized to using grid resources.

The Grid Security Infrastructure (GSI) in WLCG/EGEE enables secure authentication and communication over an open network. GSI is based on public key encryption, X.509 certificates, and the Secure Sockets Layer (SSL) communication protocol, with extensions for single sign-on and delegation.

In order to authenticate himself, a user needs to have a digital X.509 certificate issued by a Certification Authority (CA) trusted by the infrastructure running the middleware.

The authorisation of a user on a specific Grid resource can be done in two different ways. The first is simpler, and relies on the grid-mapfile mechanism. The second way relies on the Virtual Organisation Membership Service (VOMS) and the LCAS/LCMAPS mechanism, which allow for a more detailed definition of user privileges.

User interface

The access point to the gLite Grid is the User Interface (UI). This can be any machine where users have a personal account and where their user certificate is installed. From a UI, a user can be authenticated and authorized to use the WLCG/EGEE resources, and can access the functionalities offered by the Information, Workload and Data management systems. It provides CLI tools to perform some basic Grid operations:

- list all the resources suitable to execute a given job;
- submit jobs for execution;
- cancel jobs;
- retrieve the output of finished jobs;
- show the status of submitted jobs;
- retrieve the logging and bookkeeping information of jobs;
- copy, replicate and delete files from the Grid;
- retrieve the status of different resources from the Information System.

Computing element

A Computing Element (CE), in Grid terminology, is some set of computing resources localized at a site (i.e. a cluster, a computing farm). A CE includes a Grid Gate (GG)1, which acts as a generic interface to the cluster; a Local Resource Management System (LRMS) (sometimes called batch system), and the cluster itself, a collection of Worker Nodes (WNs), the nodes where the jobs are run.

There are two GG implementations in gLite 3.1: the LCG CE, developed by EDG and used in LCG-22, and the gLite CE, developed by EGEE. Sites can choose what to install, and some of them provide both types. The GG is responsible for accepting jobs and dispatching them for execution on the WNs via the LRMS.

In gLite 3.1 the supported LRMS types are OpenPBS/PBSPro, Platform LSF, Maui/Torque, BQS and Condor, and Sun Grid Engine.

Storage element

A Storage Element (SE) provides uniform access to data storage resources. The Storage Element may control simple disk servers, large disk arrays or tape-based Mass Storage Systems (MSS). Most WLCG/EGEE sites provide at least one SE.

Storage Elements can support different data access protocols and interfaces. Simply speaking, GSIFTP (a GSI-secure FTP) is the protocol for whole-file transfers, while local and remote file access is performed using RFIO or gsidcap.

Most storage resources are managed by a Storage Resource Manager (SRM), a middleware service providing capabilities like transparent file migration from disk to tape, file pinning, space reservation, etc. However, different SEs may support different versions of the SRM protocol and the capabilities can vary.

There is a number of SRM implementations in use, with varying capabilities. The Disk Pool Manager (DPM) is used for fairly small SEs with disk-based storage only, while CASTOR is designed to manage large-scale MSS, with front-end disks and back-end tape storage. dCache is targeted at both MSS and large-scale disk array storage systems. Other SRM implementations are in development, and the SRM protocol specification itself is also evolving.

Classic SEs, which do not have an SRM interface, provide a simple disk-based storage model. They are in the process of being phased out.

Information service

The Information Service (IS) provides information about the WLCG/EGEE Grid resources and their status. This information is essential for the operation of the whole Grid, as it is via the IS that resources are discovered. The published information is also used for monitoring and accounting purposes.

Much of the data published to the IS conforms to the GLUE Schema, which defines a common conceptual data model to be used for Grid resource monitoring and discovery.

The Information System that is used in gLite 3.1 inherits its main concepts from the Globus Monitoring and Discovery Service (MDS). However, the GRIS and GIIS in MDS has been replaced by the Berkeley Database Information Index which is essentially an OpenLDAP server that is updated by an external process.

Workload management

The purpose of the Workload Management System (WMS) is to accept user jobs, to assign them to the most appropriate Computing Element, to record their status and retrieve their output. The Resource Broker (RB) is the machine where the WMS services run.

Jobs to be submitted are described using the Job Description Language (JDL), which specifies, for example, which executable to run and its parameters, files to be moved to and from the Worker Node on which the job is run, input Grid files needed, and any requirements on the CE and the Worker Node.

The choice of CE to which the job is sent is made in a process called match-making, which first selects, among all available CEs, those which fulfill the requirements expressed by the user and which are close to specified input Grid files. It then chooses the CE with the highest rank, a quantity derived from the CE status information which expresses the *goodness* of a CE (typically a function of the numbers of running and queued jobs).

The RB locates the Grid input files specified in the job description using a service called the Data Location Interface (DLI), which provides a generic interface to a file catalogue. In this way, the Resource Broker can talk to file catalogues other than LFC (provided that they have a DLI interface).

The most recent implementation of the WMS from EGEE allows not only the submission of single jobs, but also collections of jobs (possibly with dependencies between them) in a much more efficient way then the old LCG-2 WMS, and has many other new options.

Finally, the Logging and Bookkeeping service (LB) tracks jobs managed by the WMS. It collects events from many WMS components and records the status and history of the job.

Software components

Below you can find a list of the current gLite components and services that are deployed or about to be deployed on the production infrastructure, together with the contributing partners (from the JRA1 home page):

- VOMS and VOMSAdmin (INFN)
- Proxy and attribute certificate renewal (CESNET)
- Shibboleth interoperability: SLCS, VASH, STS (SWITCH)
- LCAS/LCMAPS (NIKHEF)
- gLExec (NIKHEF)
- Delegation Framework (CERN, HIP, STFC)
- CGSI_gSOAP (CERN)
- gsoap-plugin (CESNET)
- Trustmanager (HIP)
- Util-java (HIP)
- Gridsite (STFC)
- Authorization Framework (HIP, INFN, NIKHEF, SWITCH)
- BDII (CERN)
- Grid Laboratory Uniform Environment (CERN)
- R-GMA (STFC)
- CREAM (INFN)
- CEMon (INFN)
- BLAH (INFN)
- WMS (INFN, ElsagDatamat)
- LB (CESNET)

- DPM (CERN)
- GFAL (CERN)
- LFC (CERN)
- FTS (CERN)
- lcg_utils (CERN)
- EDS and Hydra (HIP)
- AMGA (CERN, KISTI, INFN)

Please note that this is not a definitive list of all components.

Flow (software)

Flow is middleware software, which allows data integration specialists to connect disparate systems, whether they are on-premise, hosted or in the cloud; transforming and restructuring data as required between environments. Flow functionality can be utilised for data integration projects, EDI and data conversion activities. Flow has been created by Flow Software Ltd in NZ and is available through a variety of partner companies or directly from Flow Software in NZ and Australia.

Integration software allows organisations to continue using existing applications, overcoming the need to customize or upgrade as their requirements change. By using integration software, many businesses benefit from reduced dependence on manual keying of data and the avoidance of costs and delays caused by keying errors.

Flow Software Features

Flow enables data management:

- Transformation of data, within and between sets
- Generation and consumption of data, accessioning from specified sets within structures
- Transportation of data files, using various transport formats, including secure
- Specification of task work-flows
- Notification of transactions and formats via reports

Data Generation

Flow accesses and generates data in structured formats, from files or databases.

Flow can access and read from, or write to databases using either the SQL89 or SQL92 specification. Informix provides support for extended SQL use.

- Microsoft SQL Server 2000 & above

- Microsoft Access 97 97 above
- MySQL 4.x
- Oracle 8i
- InterBase 5.6
- Informix
- IBM DB2
- MYOB
- Any ODBC compliant database as per the Microsoft ODBC specification
- Any ADO compliant data source as per the Microsoft ADO specification

Flow can access and read from, or write to various file types.

- Any ASCII format file
- Any EDI type file based on either UN/EDIFACT or ANSI X12 standards
- Any XML file based on XML standards such as SOAP, XHTML or ebXML

Data Transformation

A visual mapping engine is used to configure data transformation between data sets. Data can be restructured as it is transformed, thus allowing for dissimilar data structures between source and destination. Flow data access operates independently of the mapping layer. The applied mapping logic uses events containing Object Pascal code.

Data Transportation

Flow transports generated data and files using the following formats:

- Local file access
- LAN
- FTP
- HTTP/SPOST
- HTTP/S GET
- SMTP
- POP
- ebMS
- SOAP

User Interface

The Flow user interface allows users to create and processes, activate processes and view activity logs.

Email notifications of Flow process activity can also be configured.

Actions

Flow uses predefined processing of events that can be executed either on schedule, or event driven. Actions and their results are logged and available via the user interface.

Actions include:

- Transformation of data or files
- Generation of specific reports
- Windows-based shell commands
- Outward-bound transports of data or files
- Selected SQL statements
- Custom plugin actions

Reports

Flow includes a report writer based on the software Report Builder. The report writer can create custom notification reports providing users with details related to their transactions. Reports can be created in XML, PDF, JPEG and XLS. Reports can be embedded into email messages if required.

History

Flow was created and developed by company founder Cameron Hart in North Shore City, New Zealand.

Chapter-3

Advanced Message Queuing Protocol

The **Advanced Message Queuing Protocol** (**AMQP**) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP mandates the behaviour of the messaging provider and client to the extent that implementations from different vendors are truly interoperable, in the same way as SMTP, HTTP, FTP, etc. have created interoperable systems. Previous attempts to standardise middleware have happened at the API level (e.g. JMS) and this did not create interoperability. Unlike JMS, which merely defines an API, AMQP is a wire-level protocol. A wire-level protocol is a description of the format of the data that is sent across the network as a stream of octets. Consequently any tool that can create and interpret messages that conform to this data format can interoperate with any other compliant tool irrespective of implementation language.

Overview

AMQP was originally designed to provide a vendor-neutral (i.e. interoperable across multiple vendors) protocol for managing the flow of messages across an enterprise's business systems.

AMQP is middleware to provide a point of rendezvous between backend systems, such as data stores and services and front end systems such as end user applications. The first applications happen to have been in the financial industry, i.e. trading desks, where real time order and market data are transmitted. Though originally used inside of enterprises, AMQP can easily be used to move messages between organizations.

AMQP lets system architects build common messaging patterns out of a simpler underlying model. Typical messaging patterns are: *request-response*, in which messages are sent to or from specific recipients, *publish-and-subscribe*, in which information is distributed to a set of recipients according to various subscription criteria, and *round-robin*, in which tasks are distributed fairly among a set of recipients. Realistic

applications combine these, e.g. round-robin for distributing work plus request-response for sending back responses.

The protocol specification defines a binary wire protocol used between a *client* and *server* (also known as a *broker*). In addition the specification outlines a messaging queuing model and services that an implementation provides.

The queuing model of AMQP provides for a wide range of messaging use-cases and further refines the functions of the clients and brokers. The function of brokers can be usefully broken into two kinds: exchanges and message queues. Message queues store messages, and various implementations can achieve various quality of service. For example a slow but tornado-proof message queue would keep redundant copies in multiple geographic regions while a fast but fragile message queue might keep everything in a single process's RAM. To help improve interoperability some of these aspects of the message queues are specified in the protocol, e.g. you can state what you need asking a message queue implementing broker to create a new queue.

The standard AMQP exchanges have no semantics for storing messages. They route them to queues, which store them on behalf of recipients. Exchanges implement a range of message routing techniques: one-to-one message passing (like email to one recipient), one-to-N (like an email list), one-to-one-of-N (like a queue for the next open checkout), and so on. Since all exchanges accept messages from N senders, AMQP allows all one-to-any routing to be N-to-any. The rules that configure an exchange, known as *bindings*, can range from very simple (pass everything into this message queue) to procedural inspections of message content. AMQP allows arbitrary exchange semantics through *custom exchanges* (which can queue, generate, consume, and route messages in any way desired by the implementation).

Messages consist of an envelope of properties used in routing and by applications and a content, of any size. AMQP message contents are opaque binary blobs. Messages are passed between brokers and clients using the protocol commands Basic.Publish and Basic.Deliver. These commands are asynchronous so that conditions that arise from a command's evaluation are signalled by sending additional commands back on the channel that carried the command originally. AMQP also provides a synchronous message delivery command, Basic.Get/Get-Ok.

Examples of error conditions include signalling by an exchange that it could not route a message because no route was found, or signalling that a message queue declined to accept a message (say because it was full). Message brokers may be configured to handle exceptions in different ways. For example, routing the associated message to a dead letter queue or even bringing the broker to a hard stop.

Development

AMQP was developed from mid-2004 to mid-2006 by JPMorgan Chase & Co. and iMatix Corporation who also developed implementations in C/C++ and Java. JPMorgan

Chase & Co. and iMatix documented the protocol as an interoperable specification and assigned to a working group that included Red Hat, Cisco Systems, TWIST, IONA, and iMatix. As of November 2009, the working group consists of Bank of America, Barclays, Cisco Systems, Credit Suisse, Deutsche Börse Systems, Envoy Technologies, Inc., Goldman Sachs, Progress Software, iMatix Corporation, JPMorgan Chase Bank Inc. N.A, Microsoft Corporation, Novell, Rabbit Technologies Ltd., Red Hat, Inc., Solace Systems, Tervela Inc., TWIST Process Innovations ltd, WS02 and 29West Inc.

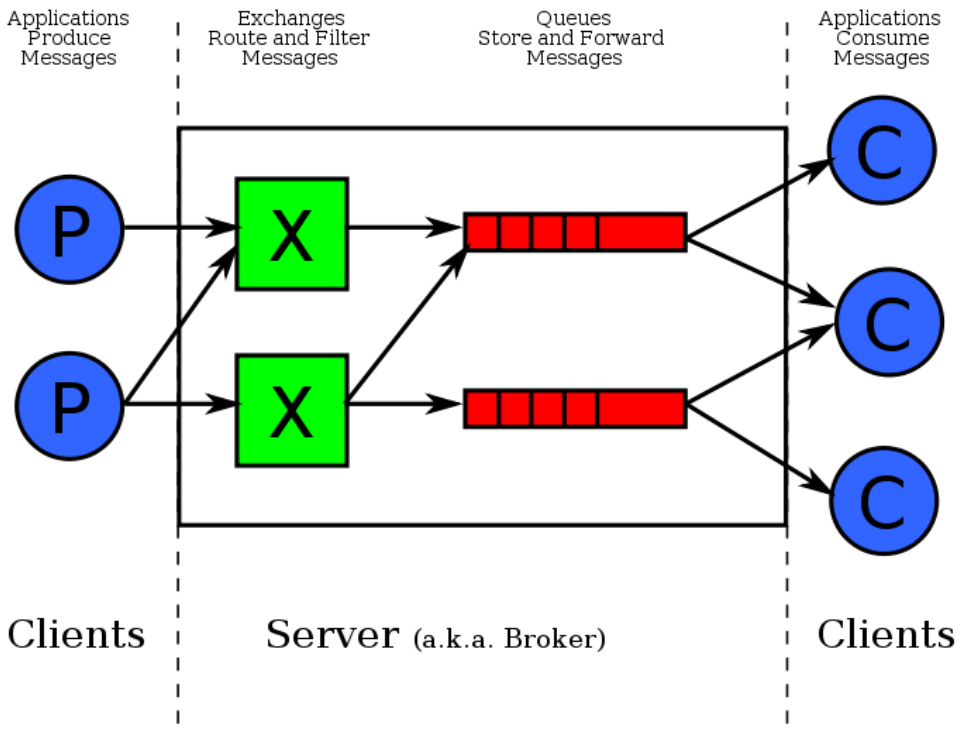
A notable design goal of AMQP was to enable the creation of open standard protocol stacks for business messaging both within and between firms by combining AMQP with one of the many open standards describing business transactions, such as FpML or more generically as a reliable transport for SOAP.

Whilst AMQP originated in the financial services industry, it has general applicability to a broad range of middleware problems.

The AMQP model

AMQP defines a number of entities. From a connection perspective the relevant ones are:

- Message broker: a server to which AMQ clients connect using the AMQ protocol. Message brokers can run in a cluster but these details are implementation specific and are not covered by the specification.
- User: a user is an entity that, by providing credentials in form of a password, may or may not be authorized to connect to a broker.
- Connection: a physical connection e.g. using TCP/IP or SCTP. A connection is bound to a user.
- Channel: a logical connection that is tied to a connection. Hence communication over a channel is stateful. Clients that perform concurrent operations on a connection should maintain a distinct channel for each of those. Clients that use a threaded model of concurrency can for example encapsulate the channel declaration in a thread-local variable.



Entities in the AMQP model used for message transfer

The entities used for the actual sending and receiving of messages are all declared on a channel. A declaration assures the issuing client that the entity exists (or was previously declared by another client). Any attempt to declare a named entity with different properties than it was declared before will result in an error. In order to change the properties of such an entity it must be deleted prior to a re-declaration (with changed properties).

Some of these entities are named. The naming must be unique within the scope of the entity and its broker. Since clients usually (at least no such operations are defined in the AMQP specification) do not have the means to get a list of all available named entities, the knowledge of an entity name is what allows the client to perform operations on it.

Names are encoded in UTF-8, must be between 1 and 255 characters in length and must start with a digit, a letter or an underscore character.

Exchanges

Exchanges are the entities to which messages are sent. They are named and have a type as well as properties such as:

- passive: the exchange will not get declared but an error will be thrown if it does not exist.
- durable: the exchange will survive a broker restart.
- auto-delete: the exchange will get deleted as soon as there are no more queues bound to it. Exchanges to which queues have never been bound will never get auto deleted.

Queues

Queues are the entities which receive messages. They are named and have properties but not a type. Clients can subscribe to queues to the effect that the message broker delivers (pushes) the contents of the queue to the client. Alternatively clients can pop (pull) messages from the queue as they see fit.

Messages are guaranteed to be delivered in the order that they were first delivered to a queue, unless certain kinds of rerouting operations (e.g. due to failures) occur.

The properties of queues are:

- alternate-exchange: when messages are rejected by a subscriber or orphaned by queue deletion, its messages get routed to this exchange and get removed from the queue.
- passive: the queue will not get declared but an error will be thrown if it does not exist.
- durable: the queue will survive a broker restart.
- exclusive: there can only be one client for this specific queue.
- auto-delete: the queue will get deleted as soon as no more subscriptions are active on it. This shares the same constraint as the auto-delete property for exchanges: if no subscription has been ever active on the queue it will not get auto-deleted. An exclusive queue however will always get auto-deleted when the client terminates its session.

Note that queues are scheduled to replace exchanges in AMQP/1.0.

Messages

Messages are unnamed and are published to an exchange. They consist of a header and a content body. While the body is opaque data the header contains a number of optional properties:

- routing-key: this field is used in ways dependent on the type of the exchange.
- immediate: the message will get handled as unroutable if at least one of the queues which would receive the message has no subscription on it.
- delivery-mode: indicates that a message might need persistence. Only for such messages the broker makes a best-effort to prevent a loss of the message before consumption. If there is uncertainty on the broker's end about the successful

delivery of a message (e.g. in case of errors) it might deliver a message more than once. Non persistent delivery modes do not show this kind of behavior.
- priority: an indicator (a range between 0 and 9) that a message has higher precedence than others.
- expiration: the duration in milliseconds before the broker may handle the message as unroutable.

Bindings

A binding is a relationship between one queue and one exchange that specifies how messages flow from the exchange to the queue. The binding properties match the routing algorithm used in exchanges. Bindings (and exchange algorithms) can be placed on a curve of increasing complexity:

- Unconditional - the binding has no properties and requests "all" messages from the exchange.
- Conditional on a fixed string - the binding has one property, the **routing key** and requests all messages that have an identical routing key.
- Conditional on a pattern match - the binding has one property, the **routing key** and requests all messages that match the routing key using a pattern-matching algorithm. Arbitrary pattern syntaxes could be used. AMQP implements topic matching.
- Conditional on multiple fixed strings - the binding has a table of properties, the **arguments** and requests all messages whose headers match these arguments, using logical ANDs or ORs to combine matches.
- Conditional on multiple patterns - the binding has a table of properties, the **arguments** and requests all messages whose headers match these arguments, using a pattern matching algorithm and logical combinations.
- Conditional on algorithmic comparison - the binding has an algorithmic expression (like an SQL SELECT WHERE clause) and requests all messages whose headers match that expression.
- Conditional on content inspection - the binding specifies arbitrary criteria that are resolved by inspection of the actual message content.

Not all these are implemented as standard, or by all implementations.

Exchange types and the effect of bindings

These four entities form the basic model of the AMQP. The key to understand how a message is passed to a queue lies in the relationship between the type of an exchange and the resulting interpretation of the routing key.

An exchange will deliver up to one copy of a message to a queue if the routing key in the message matches a binding (subsequent semantically identical bindings will not lead to duplicate copies). What constitutes a match however is solely dependent on the type of an exchange:

- a direct exchange matches when the routing key property of a message and the key of the binding are identical.
- a fanout exchange always matches, even on bindings without a key.
- a topic exchange matches the routing key property of a message on binding key words. Words are strings which are separated by dots. Two additional characters are also valid: the *, which matches 1 word and the #, which matches 0..N words. Example: *.*stock.*# matches the routing keys *usd.stock* and *eur.stock.db* but not *stock.nasdaq*.
- a headers exchange matches on the presence of keys as well as key–value pairs which can be concatenated with logical and–or connections in a messages header. In this case the routing key is not a criterion for matching that is considered by the exchange. Neither does the binding carry a single routing key but special format which contains header keys and / or key-value-pairs which match on the header key being present or the header key being present and the value being the same respectively.

Other e.g. vendor-specific exchanges are explicitly permitted in the specification.

The concept of binding named queues to named exchanges has powerful properties (with binding making those two entities independent of each other). It is, for instance, possible to bind a single queue with multiple bindings to the same or to different exchanges. Or multiple consumers can share the name of a queue and bind to it with the same parameters and will therefore get only message that the other consumers did not consume. Or multiple consumers can declare independent queues but share the bindings and get all the message every other consumer would get on the bound exchange with these bindings.

Specification revisions and the future of AMQP

The following specifications of the AMQ protocol have been published, in chronological order:

- 0-8 in June 2006
- 0-9 in December 2006
- 0-10 (documents are undated)
- 0-9-1 in November 2008
- 1.0 draft in May 2010

The draft 1.0 specification changes the AMQP model illustrated above by removing the concepts of exchanges and bindings, and replacing these with queues and *links*. This change aims to remedy two problems with the previous approach:

1. The publisher needs to know too much about the receivers topology (what exchanges and exchange types are available).

2. Producer flow control is challenging - if an Exchange is routing a message to 2 different queues, one empty and the other nearly full, what flow control information should be relayed to the producer and how would that be determined?

According to John O'Hara however, JPMorganChase and RedHat introduced links into AMQP/1.0 simply to solve an operational problem of slow consumers causing memory build up in brokers.

Other changes include the introduction of a queue addressing schema similar to E-mail and XMPP. This raises addresses to first-class entities, and allows for the publication of service location records using the DNS.

The process of bringing the 1.0 Specification to a Standard involves a requirement elicitation phase, then the release of a "public review" spec (PR) which should be reviewed and asked for comments, optionally resulting in further modifications. When there are no substantive changes to the PR, it is voted to be the 1.0 Recommendation. When there are at least two implementations that pass a special test coverage, the Recommendation is voted to be 1.0 Standard. As of 29-Dec-2010, a Recommendation spec has been produced and is waiting for two or more implementations proven to interoperate.

Implementations

These are the known publicly available AMQP implementations:

- OpenAMQ, an open-source implementation of AMQP, written in C by iMatix. Runs on Linux, AIX, Solaris, Windows, OpenVMS. APIs in C/C++ and Java JMS. Discontinued by iMatix after their switching to ØMQ.
- StormMQ, currently the only message queuing service using AMQP. Offered as a managed service, there is no software to install and no licence fees to pay. Customers pay for a subscription out of OPEX, and use a solution in the Cloud, Co-Hosted or On-Site.
- RabbitMQ, an independent open-source implementation bought by VMware in 2010. The server is written in Erlang.
- Apache Qpid, a project in the Apache Foundation. Bindings to many languages without the use of DLLs.
- Red Hat Enterprise MRG implements the latest version of AMQP 0-10 providing rich set of features like full management, federation, Active-Active clustering using Apache Qpid as upstream, adds a web console and many enterprise features. Also available in the latest 3 versions of Fedora as AMQP Infrastructure.

Other Products Integrating with AMQP

There are many integrations of other products with AMQP, including:

- Zyre, a broker that implements RestMS and AMQP to provide RESTful HTTP access to AMQP networks.

Clients

There are many clients, including:

- DE.SETF.AMQP, a Common Lisp client library for AMQP.

Comparative specifications

These are the known open specifications that cover the same or similar space as AMQP:

- Stomp, a text-based pub-sub protocol developed at Codehaus; uses the JMS-like semantics of 'destination'.
- RestMS, an HTTP-based message routing and queuing protocol that provides AMQP interoperability through an optional profile.
- XMPP, the Extensible Messaging and Presence Protocol.

There are also vendor specific, proprietary specifications includes those by the Amazon Simple Queue Service, IBM WebSphere MQ, Microsoft Message Queuing, JMS and the OpenWire as used by ActiveMQ

There has not as yet been a formal comparison of these and other protocols in the same space, although an informal comparison of XMPP and AMQP may be found here. JMS, the Java Messaging service, is often compared to AMQP. However, JMS is an API specification (part of the Java EE specification) that defines how message producers and consumers are implemented. JMS does not guarantee interoperability between implementations, and the JMS-compliant messaging system in use may need to be deployed on both client and server. On the other hand, AMQP is a wire-level protocol specification. In theory AMQP provides interoperability as different AMQP-compliant software can be deployed on the client and server sides. Note that, like HTTP and XMPP, AMQP does not have a standard API.

Chapter-4

IBM WebSphere Message Broker and HP RTR

IBM WebSphere Message Broker

WebSphere Message Broker (WMB) is IBM's information broker from the WebSphere product family that allows business information to flow between disparate applications across multiple hardware and software platforms. Business rules can be applied to the data flowing through the message broker to route and transform the information. The product can be considered to be an Enterprise Service Bus providing connectivity between disparate applications.

History

Originally the product was developed by a company named 'NEON', short for 'New Era of Networks' which was later acquired by Sybase. The product was developed by New Era of Networks and re-branded as IBM product called 'MQSeries Integrator' or 'MQSI' for short. Versions of MQSI ran up to 2.1. The product was added to the WebSphere family and rebranded 'WebSphere MQ Integrator', still version 2.1. From 2.1 the version numbers are synchronized with the rest of the WebSphere family and jumped to version 5.0. The name changed to 'WebSphere Business Integration Message Broker' (WBIMB). In this version the development environment was redesigned using Eclipse and support for Web services was integrated into the product. Since V6.0 the product has been known as 'WebSphere Message Broker'. WebSphere Message Broker version 7.0 was announced in October 2009.

Components

WebSphere Message Broker consists of the following components:

- WebSphere Message Broker runtime

- WebSphere Message Broker Toolkit
- WebSphere Message Broker Explorer

How Message Broker works

Overview

The WebSphere Message Broker Toolkit enables developers to graphically design message flows and related artifacts. Once developed, these resources can be packaged into a broker archive (BAR) file and deployed into the runtime environment. At this point, the broker is able to continually process messages according to the logic described by the message flow.

WebSphere Message Broker flows can be used in a Service Oriented Architecture, and if properly designed by Middleware Analysts, integrated into event-driven SOA schemas, sometimes referred to as SOA 2.0.

The workflow a developer would perform to create WMB functionality is cyclical, and probably more agile than most other software development. Developers will create a message flow, generate a BAR file, deploy the message flow contained in the BAR file, test the message flow and repeat as necessary to achieve reliable functionality.

Expected Performance

Performance varies by platform and application. Reference implementations have been bench tested to achieve transaction rates as high as 10,000 TPS. The best performing WMB system resides on AIX Power7 architecture due to its throughput characteristics, very large memory blocks of contiguous memory, 32 MB of L3 cache, 8 terabytes of local address space, 2 petabytes of global address space and 256 logical CPUs per system. Among the slower performers for WMB runtime bench tests are z/OS implementations because of the lack of very large contiguous memory block allocations within the OS. Whereas on AIX Power7, 10,000 TPS is attainable, on z/OS 400 TPS is more likely.

Cost Per Transaction

WMB price per transaction on AIX Power7 considering three-year total cost of acquisition of hardware, software, maintenance is scalable much more economically than other platforms. Some people would call this "bang-for-the-buck", value delivery, or Return-On-Investment (ROI). This means that counting all costs involved, not necessarily the original purchase price, the Power7 architecture can be much more economically viable than other offerings, if implemented with the right expertise. Because an AIX Power7 WMB application can achieve a sustained 10,000 TPS throughput, in the course of one day, 864 million transactions can be serviced (10,000 * 60 seconds * 60 minutes * 24 hours). In one year, with ten percent downtime for maintenance, this equates to 283 billion transactions. In three years, this is 849 billion transactions. Cost per transaction

then could be as little as 0.25 cents (four transactions per penny) given a three year total cost of ownership around US$200 million. Using a real-life example, when compared to the price being charged merchants by some credit card networks, of sixty-four cents per transaction or greater, four transactions for a penny seems pretty sensible.

Broker nodes available

A developer can choose from many pre-designed broker nodes. Nodes have different purposes. Some nodes map data from one format to another (for instance, Cobol or PL/I Copybook to canonical XML). Other nodes evaluate content of data and route the flow differently based on certain criteria.

Node types

There are many types of node that can be used in developing message flows; the following node transformation technology options are available:

- Extended Structured Query Language (ESQL)
- Graphical Message Mapping
- eXtensible Stylesheet Language Transformations (XSLT)
- JavaCompute (as of version 6)
- WebSphere Transformation Extender (formerly known as Ascential DataStage TX, DataStage TX and Mercator Integration Broker) is available as a separate licensing option
- PhpCompute (as of version 6.1.0.3)

Patterns

A pattern captures a commonly recurring solution to a problem (example: Request-Reply pattern). The specification of a pattern describes the problem being addressed, why the problem is important, and any constraints on the solution. Patterns typically emerge from common usage and the application of a particular product or technology. A pattern can be used to generate customized solutions to a recurring problem in an efficient way. We can do this pattern recognition or development through a process called service oriented modeling.

WebSphere Message Broker version 7 introduced patterns that:

- Give you guidance in implementing solutions
- Increase development efficiency because resources are generated from a set of predefined templates
- Improve quality through asset reuse and common implementation of functions such as error handling and logging

The patterns cover a range of categories including file processing, application integration, and message based integration.

Pattern Examples

- Fire-and-Forget (FaF)
- Request-Reply (RR)
- Aggregation (Ag)
- Sequential (Seq)

Supported platforms

Operating systems

Currently available platforms for WebSphere Message Broker are:

- AIX
- HP-UX (IA64)
- Solaris (SPARC and x86-64)
- Linux (x86, x86-64, PPC and 390x)
- Microsoft Windows
- z/OS

Trivia

- The Configuration Manager repository was named "BERNARD" after the Configuration Manager development team's pet gargoyle.

HP RTR

HP Reliable Transaction Router (RTR) is a transactional middleware and is simply referred to as RTR by its users. RTR is used to integrate with applications that require reliable transaction services. RTR is currently available on HP-UX, Linux, Windows and OpenVMS.

Reliable Transaction Support

RTR manages the messages sent between client-server to provide node and network fail-over for increased reliability, transactional integrity, and interoperability between dissimilar systems. RTR does this with a scalable, easy-to-use design.

Three-Tier Architecture

The RTR software has three logical entities and referred to as Front-End (FE), Back-End (BE) and Transaction-Router(TR). The router is a software component that provides the fail-over intelligence and manages connections to the Back-End. The client applications

running on the Front-End combined with Router and Server applications running on Back-End interact to provide transaction integrity and reliability. The three logical entities can exist on the same node but are usually deployed on different nodes to achieve modularity, scalability and high availability.

The client application interacts with the Front-End which forwards the messages to the Router, the Router in turn routes the message to the intended Back-End where the appropriate Server application is available for processing the message.

Message Content Routing

The RTR routing capability is that it allows to partition data across multiple servers and nodes for increased performance. Within an application, the partition determines how messages are routed between the client and the servers.

Transactional Messaging

The message exchange happens between the client and server. Transactions start at the client and involve many messages that can go to a number of different servers.

Broadcast Messaging

Such method of messaging is used in situations where there are multiple recipients for a message, or where unsolicited messages need to be sent.

Replication

RTR can help survive the failures generally seen in distributed application environment which include complete site failure, node failure, network link failure and software process failure. RTR also provides continuous availability by using redundant resources in the distributed environment.

RTR Environment

RTR provides a **Web Interface** and a **Command Line Interface**(CLI) for managing the RTR environment. When RTR and its components are running along with the applications, then Client Application, Server Application, RTR services will be active.

RTR Features

RTR is widely used as both a product which is integrated with client applications and as well as integrated in an infrastructure developed and customized to suit various user needs. It is widely popular amongst its users for its seamless integration into the user's solutions.

APIs

User and Management Applications can be written using RTR APIs. The C, C++, Java and .Net variants of APIs are available for creating applications to use RTR.

Compatibility Matrix

RTR is currently supported on the Linux, Windows and OpenVMS platforms.

Chapter-5

Internet Communications Engine and Volunteer Computing

Internet Communications Engine

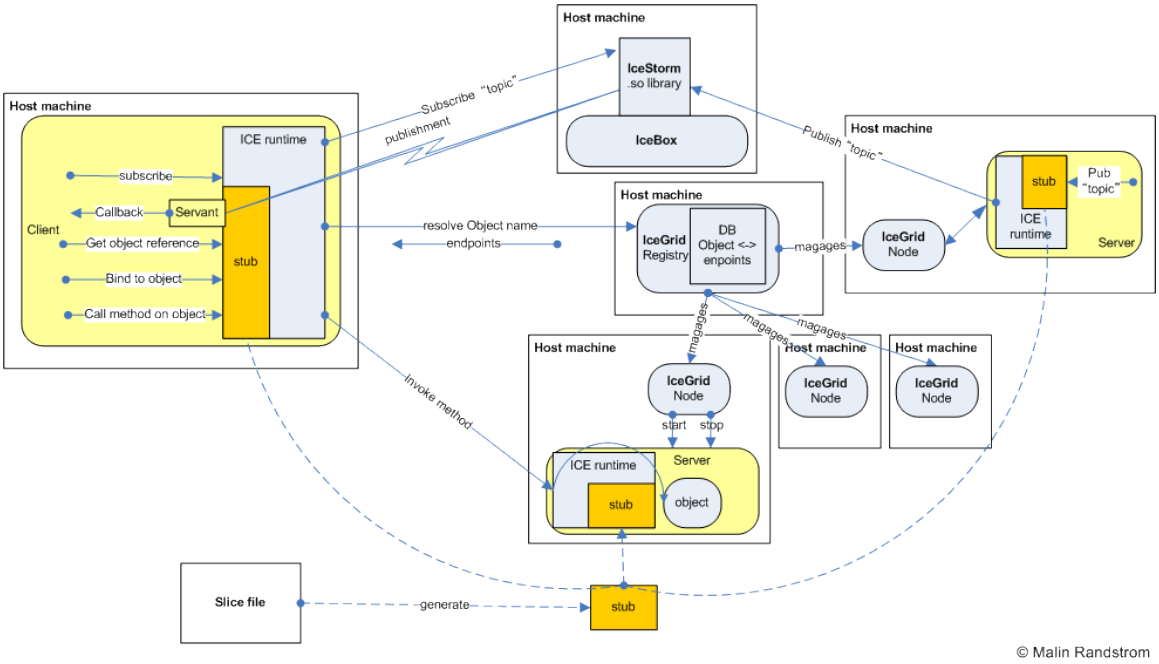
The **Internet Communications Engine**, or **Ice**, is an object-oriented middleware that provides object-oriented Remote Procedure Call, grid computing, and Publish/subscribe functionality developed by ZeroC and dual-licensed under the GNU GPL and a proprietary license. It supports C++, Java, .NET-languages (such as C# or Visual Basic), Objective-C, Python, PHP, and Ruby on most major operating systems such as Linux, Solaris, Windows and Mac OS X. A light variant of ICE runtime, called Ice-e, may run inside mobile phones. As its name indicates, the middleware may be used for internet applications without the need to use the HTTP protocol and is capable of traversing firewalls unlike most other middleware.

ICE and CORBA

ICE was influenced by CORBA in its design, and indeed was created by several influential CORBA developers, including Michi Henning. However, it is much smaller and less complex than CORBA. According to ZeroC's webpages, this is partly a result of being designed by a small group of experienced developers, instead of suffering from design by committee.

ICE Components

ICE is a set of CORBA like components that include object-oriented remote-object-invocation, replication, grid-computing, failover, load-balancing, firewall-traversals, and publish-subscribe services. To gain access to those services, applications are linked to a stub library or assembly, which is generated from a language-independent IDL-like syntax called *slice*.



IceStorm

is an object-oriented publish-and-subscribe framework that also supports federation and quality-of-service. Unlike other publish-subscribe frameworks such as TIBCOs' Rendezvous or SmartSockets, message content consist of objects of well defined classes rather than of structured text.

IceGrid

is a suite of frameworks that provide object-oriented load balancing, failover, object-discovery and registry services.

IcePatch

facilitates the deployment of ICE based software. For example, a user who wishes to deploy new functionality and/or patches to several servers may use IcePatch.

Glacier

is a proxy-based service to enable communication through firewalls, thus making ICE an internet communication engine.

IceBox

is a SOA-like container of executable services implemented in .dll or .so libraries. This is a lighter alternative to building entire executable for every service.

Slice

Slice is a Zeroc-proprietary file format that programmers follow to edit computer-language independent declarations and definitions of classes, interfaces, structures and enumerations. Slice definition files are used as input to the stub generating process. The stub in turn is linked to applications and servers that should communicate with one another based on interfaces and classes as declared/defined by the slice definitions.

Apart from CORBA, classes and interfaces support inheritance and abstract classes. In addition, slice provides configuration options in form of macros and attributes to direct the code generation process. An example is the directive to generate a certain STL `list<double>` template instead of the default, which is to generate a STL `vector<double>` template.

Comparisons to other major middleware

SOAP

Ice also compares favorably to SOAP, the main advantages being that it's more object oriented, and offers vastly superior performance in terms of both bandwidth and processor load, because SOAP is based on HTTP and XML, which needs to be parsed, while Ice uses a binary protocol designed for high performance and low verbosity. However, Ice might not offer similar performance or compactness advantages when SOAP messages are exchanged using a more efficient transport and message encoding such as SOAP/TCP and Fast Infoset.

CORBA

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together.

TIBCO Rendezvous/EMS

Rendezvous is an asynchronous publish/subscribe only middleware from TIBCO that provides text based messaging as well as its own proprietary name value pair format. A daemon runs at the client side and communicates with subscribing client processes through IPC pipes or TCP/IP. The daemon mediates between client processes and daemons that handle publishing servers. Those daemons support multicast as well as broadcast communication.

EMS stands for Enterprise Messaging Services and is a JMS server implementation, that also has support for TIBCO Rendezvous.

Talarian Smartsockets

The differences to Rendezvous/EMS above are the lack of a payload format and a daemon running on the client side. Instead, a number of publish/subscribe daemons run somewhere on the network collectively called a cloud. This provides better performance and failover since the communication is split between several daemons and once a daemon run into an unavailable state, clients may automatically switch to another daemon. The cloud also catches published data and provides an interface for clients to retrieve the data. Any clients may therefore request for last published data anytime without having to wait for sources to republish the data. The latter mechanism is currently missing in ICE.

Volunteer computing

Volunteer computing is a type of distributed computing in which computer owners donate their computing resources (such as processing power and storage) to one or more "projects".

History

The first volunteer computing project was the Great Internet Mersenne Prime Search, which was started in January 1996. It was followed in 1997 by distributed.net. In 1997 and 1998 several academic research projects developed Java-based systems for volunteer computing; examples include Bayanihan, Popcorn, Superweb, and Charlotte.. Another similar concept is Sideband computing which let a user to share his computing power while he is online.

The term "volunteer computing" was coined by Luis F. G. Sarmenta, the developer of Bayanihan. It is also appealing for global efforts on social responsibility, or Corporate Social Responsibility as reported in a Harvard Business Review or used in the Responsible IT forum.

In 1999 the SETI@home and Folding@home projects were launched. These projects received considerable media coverage, and each one attracted several hundred thousand volunteers.

Between 1998 and 2002, several companies were formed with business models involving volunteer computing. Examples include Popular Power, Porivo, Entropia, and United Devices.

In 2002, the Berkeley Open Infrastructure for Network Computing (BOINC) opensource project was founded, and became the software running the largest public computing grid (World Community Grid) in 2007.

Middleware for volunteer computing

The client software of the early volunteer computing projects consisted of a single program that combined the scientific computation and the distributed computing infrastructure. This monolithic architecture was inflexible; for example, it was difficult to deploy new application versions.

More recently, volunteer computing has moved to middleware systems that provide a distributed computing infrastructure independently of the scientific computation. Examples include:

- The Berkeley Open Infrastructure for Network Computing (BOINC). BOINC is the most widely-used middleware system, and is currently used by the World Community Grid. It is open source (LGPL) and is developed by an NSF-funded research project located at the UC Berkeley Space Sciences Laboratory. It offers client software for Windows, Mac OS X, Linux, and other Unix variants.
- XtremWeb is used primarily as a research tool. It is developed by a group based at the University of Paris - South.
- Xgrid is developed by Apple. Its client and server components run only on Mac OS X.
- Grid MP is a commercial middleware platform developed by United Devices and has been used in volunteer computing projects including grid.org, World Community Grid, Cell Computing, and Hikari Grid.

Most of these systems have the same basic structure: a client program runs on the volunteer's computer. It periodically contacts project-operated servers over the Internet, requesting jobs and reporting the results of completed jobs. This "pull" model is necessary because many volunteer computers are behind firewalls that don't allow incoming connections. The system keeps track of each user's "credit", a numerical measure of how much work that user's computers have done for the project.

Volunteer computing systems must deal with several problematic aspects of the volunteered computers: their heterogeneity, their churn (that is, the arrival and departure of hosts), their sporadic availability, and the need to not interfere with their performance during regular use.

In addition, volunteer computing systems must deal with several related problems related to correctness:

- Volunteers are unaccountable and essentially anonymous.
- Some volunteer computers (especially those that are overclocked) occasionally malfunction and return incorrect results.
- Some volunteers intentionally return incorrect results or claim excessive credit for results.

One common approach to these problems is "replicated computing", in which each job is performed on at least two computers. The results (and the corresponding credit) are accepted only if they agree sufficiently.

Costs for volunteer computing participants

- Increased power consumption. A CPU that is idle generally has lower power consumption than when it is active. The desire to participate may also cause the volunteer to leave the PC on overnight, or to disable power-saving features like suspend. Additionally, if adequate cooling is not in place, this constant load on the volunteer's CPU can cause it to overheat.
- Decreased performance of the PC. If the volunteer computing application attempts to run while the computer is in use, it will impact performance of the PC. This is due to increased CPU contention, CPU cache contention, disk I/O contention, and network I/O contention. If RAM is a limitation, increased disk cache misses and/or increased paging can result. Volunteer computing applications typically execute at a lower CPU scheduling priority, which helps to alleviate CPU contention.

These effects may or may not be noticeable, and even if they are noticeable, the volunteer might choose to continue participating. However the increased power consumption can be remedied to some extent by setting the option of desired processor usage percent, that is available e.g. in BOINC client.

Chapter-6

Message-oriented middleware and SynfiniWay

Message-oriented middleware

Message-oriented middleware (**MOM**) is software or hardware infrastructure focused on sending and receiving messages between distributed systems. MOM allows application modules to be distributed over heterogeneous platforms, and reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces. APIs that extend across diverse platforms and networks are typically provided by MOM.

MOM is a software that resides in both portions of client/server architecture and typically supports asynchronous calls between the client and server applications. MOM reduces the involvement of application developers with the complexity of the master-slave nature of the client/server mechanism.

Origins

A requirements story

The case of a large bank provides a good example of how middleware emerged as a business requirement:

The bank had stored all its customer details on its large mainframe since the 1960s. This mainframe remained in heavy use and underwent several upgrades.

Although ground-breaking in its day, the mainframe's usefulness to the bank’s staff diminished as the bank introduced new, separate applications based on personal computers (PCs), allowing the bank’s staff to offer customers new services that the mainframe could not support.

An ideal situation would allow the PC-based application to link to the older mainframe application and allow the mainframe and the PCs to share each other's data. Accessing the mainframe's data offers two advantages:

1. new front-end PC applications can replace the old user-unfriendly mainframe terminals
2. PC-based systems can use the data from the mainframe in new ways — previously impractical due to the constraints of the mainframe's software

Up until the late 1980s system integrators had no easy way to link these different applications together. Developers faced several challenges:

1. the developers would have to construct a separate software 'adapter' on both systems to translate data from source applications into a format that the destination system could understand (and *vice versa*).
2. the processing speed of each system would constrain the other system. For example, if the mainframe ran slowly, the PC-based application would have to wait until the mainframe caught up, thereby slowing down the PC application. Conversely, processing that had been offloaded to distributed servers for cost reasons would run slowly and the mainframe would have to wait until the server caught up.
3. communications programmers would need to install a network gateway system to form a bridge between the mainframe's network and the PC network if the different systems used different network protocols. The gateway would translate the network packets from the source system and pass them on to the destination system using the destination system's protocol.

Such issues made integration between applications difficult. Much of such integration also required re-engineering every time two applications on disparate platforms needed linking together, as every situation differed to some extent. By devoting effort to linking together applications on different systems, IT departments started to spend an amount significantly greater than that spent on original development per sub-system.

Developers needed a separate piece of software that would sit in the middle of two or more applications and would handle all the 'plumbing' between the two systems. Such software needed the intelligence to handle different platforms, different programming languages, various network protocols and diverse hardware. Developers wanted to remove themselves from the complexities of the underlying computing infrastructure so that they could focus on functionality within actual applications.

Towards the end of the 1980s middleware began to emerge that attempted to address these issues. Initial middleware offerings addressed specific handfuls of platforms or languages and thus had limited usefulness. Over time, however, middleware products have become more and more advanced, supporting multiple platforms, languages and protocols.

The ability of middleware to link together disparate systems across a heterogeneous network environment offers only one example of the benefits of this dominant technology. Middleware as of 2006 provides a whole raft of new functionality that augments and enhances the existing applications that it interconnects.

Advantages

The primary advantage of a message-based communications protocol lies in its ability to store, route or transform messages in the process of delivery.

Communication properties

Synchronicity

MOM comprises a category of inter-application communication software that generally relies on asynchronous message-passing, as opposed to a request-response metaphor. In asynchronous systems, message queues provide temporary storage when the destination program is busy or not connected. In addition, most asynchronous MOM systems provide persistent storage to back up the message queue. This means that the sender and receiver do not need to connect to the network at the same time (asynchronous delivery), and solves problems with intermittent connectivity. It also means that should the receiver application fail for any reason, the senders can continue unaffected, as the messages they send will simply accumulate in the message queue for later processing when the receiver restarts.

Routing

Many message-oriented middleware implementations depend on a message queue system. Some implementations permit routing logic to be provided by the messaging layer itself, whilst others depend on client applications to provide routing information or allow for a mix of both paradigms. Some implementations make use of broadcast or multicast distribution paradigms.

Transformation

In a message-based middleware system, the recipient's message need not replicate the sender's message exactly. A MOM system with built-in intelligence can transform messages en-route to match the requirements of the sender or of the recipient. In conjunction with the routing and broadcast/multicast facilities, one application can send a message in its own native format, and two or more other applications may each receive a copy of the message in their own native format. Many modern MOM systems provide sophisticated message transformation (or mapping) tools which allow programmers to specify transformation rules applicable to a simple GUI drag-and-drop operation.

Disadvantages

The primary disadvantage of many message oriented middleware systems is that they require an extra component in the architecture, the message transfer agent (message broker). As with any system, adding another component can lead to reductions in performance and reliability, and can also make the system as a whole more difficult and expensive to maintain.

In addition, many inter-application communications have an intrinsically synchronous aspect, with the sender specifically wanting to wait for a reply to a message before continuing. Because message-based communication inherently functions asynchronously, it may not fit well in such situations. That said, most MOM systems have facilities to group a request and a response as a single pseudo-synchronous transaction.

Lack of standards

The lack of standards governing the use of message oriented middleware has caused problems. All the major vendors have their own implementations, each with its own application programming interface (API) and management tools.

The Java EE programming environment provides a standard API called JMS (Java Message Service), which is implemented by most MOM vendors and aims to hide the particular MOM API implementations; however, JMS does not define the format of the messages that are exchanged, so JMS systems are not interoperable. Microsoft's MSMQ doesn't support JMS, although there are third-party products that can offer this. WebSphere Message Broker, from IBM, does provide JMS support, as well as a whole suite of modern functionality.

The Advanced Message Queuing Protocol (AMQP) is an emerging standard that defines the protocol and formats used in the messaging server and client, so implementations are interoperable. AMQP is defined to provide flexible routing, including common messaging paradigms like point-to-point, fanout, publish/subscribe, and request-response. It also supports transaction management, queuing, distribution, security, management, clustering, federation and heterogeneous multi-platform support. Java applications that use AMQP are typically written in Java JMS. Other implementations provide APIs for C#, C++, PHP, Python, Ruby, and other languages.

Trends

AMQP has been gaining adoption in applications that need an interoperable protocol for Message-oriented middleware.

Other protocols used for message-oriented middleware include XMPP and Streaming Text Oriented Messaging Protocol.

Message-oriented messaging protocols under development include RestMS, a protocol similar in nature to AMQP but constructed over a RESTful HTTP transport, and SPB, a minimalist message framing protocol that can be used to carry higher-level MOM protocols.

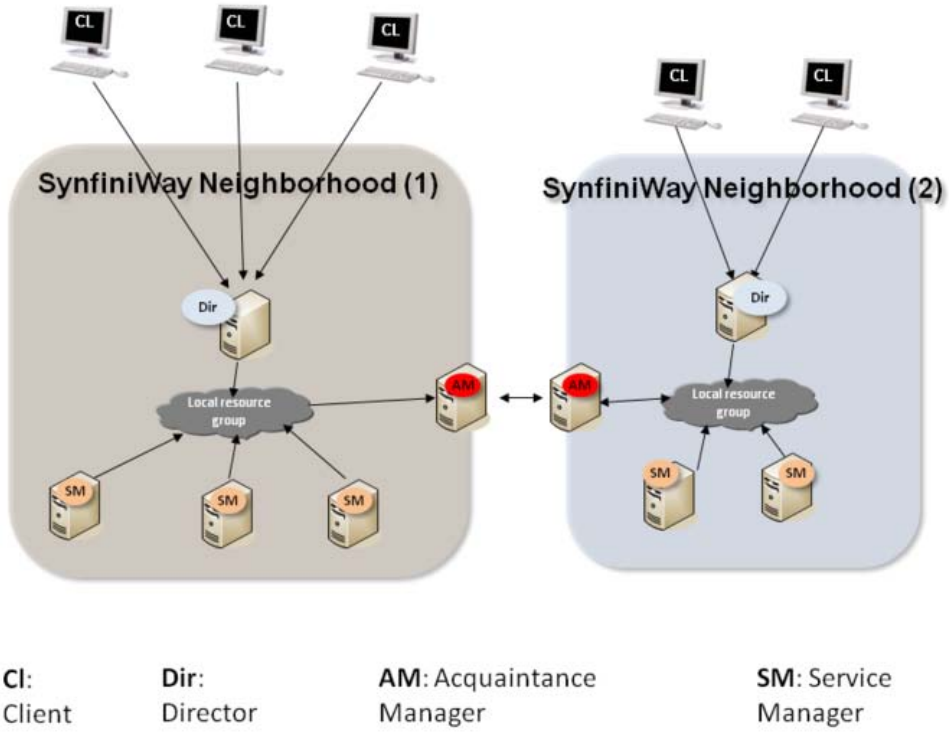
An additional trend sees message-oriented middleware functions being implemented in hardware - usually FPGAs or other specialized silicon chips.

SynfiniWay

	SynfiniWay
Developer(s)	Fujitsu
Stable release	4.0 / September 2010
Operating system	Linux, Unix, Windows
License	commercial

SynfiniWay is middleware with which a virtualised IT framework can be created that provides a uniform and global view of resources within a department, a company, or a company with its suppliers . This virtualised IT framework is service-oriented, meaning that applications are run as services, which are a system-independent view of applications. Several applications can be linked in a workflow, and data exchange between the applications participating in the workflow is implicitly managed by the IT framework. SynfiniWay is platform-independent, allowing almost any distributed heterogeneous platform to be linked into its virtualised IT framework.

IT framework



Cl: Client | **Dir**: Director | **AM**: Acquaintance Manager | **SM**: Service Manager

A **virtualised IT framework** is implemented with SynfiniWay by installing a component with specific software agents on each of the systems in the framework. There are three major types of components in SynfiniWay:

- Director, which manages end-user connection, authentication & authorisation, and workflow task scheduling and execution.
- Service Manager, which publishes and runs services on behalf of users and which executes data migration.
- Acquaintance Manager, which links one remote network, known as a SynfiniWay neighborhood, to another to allow resource discovery and file transfer between components residing in different neighbourhoods.

All components are based on Java, so that they can be deployed in a multi-platform environment. An example framework with two neighbourhoods is shown in the figure. Adding or removing components is automatically detected by the framework. The SynfiniWay meta-scheduler automatically adjusts to changes in Service Manager or service availability.

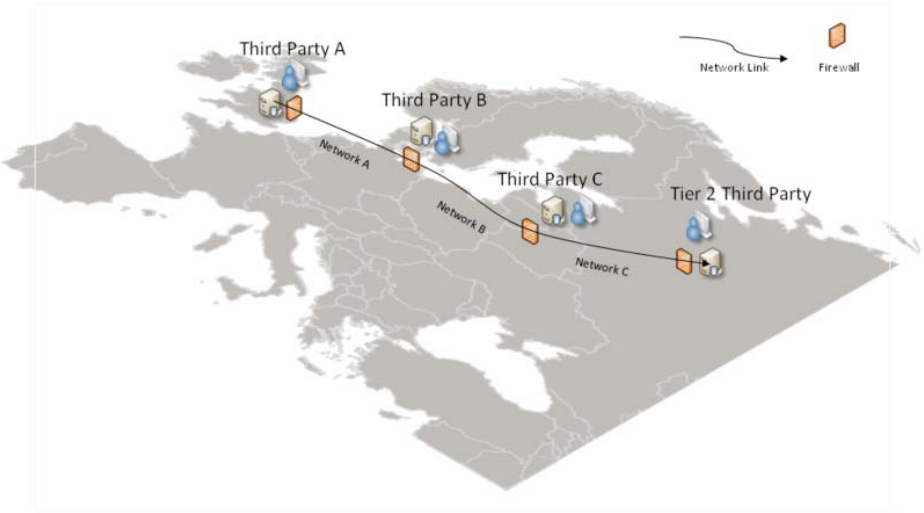
Service management

SynfiniWay is geared towards **service management** . This means that an application or a command that will be utilised is defined as a service and published on the SynfiniWay IT framework. Simple or complex tasks can be abstracted into services for execution. By using these abstracted services, a user can easily run applications or commands regardless of how complex they may be or what underlying IT infrastructure is required. They provide a form of virtualisation of computing resources since the user need not be aware of where the service is available or how it is run.

Workflow management

A technical or business process is created by linking services in a **workflow** . Workflows in SynfiniWay are based on WfMC version 1.0. A workflow defines one or more tasks that will be executed with a given execution logic (branch, loop, conditional). SynfiniWay supports multi-instance tasks which run a service multiple times concurrently. Also it supports a workflow of workflows, whereby a workflow can be executed as a task of a higher level workflow.

Data handling



Files needed by a service are automatically transferred to the computer executing that service so that the user is freed from having to manage file transfers . A file transfer mechanism is used allowing files to be transferred directly from the source to the target computer system, going through any number of firewalls between source and target, without being stored on any of the intermediary systems. This mechanism uses the shortest path for transferring files to a target computer from the source.

Meta-scheduling

SynfiniWay employs a **meta-scheduling** capability , optimizing computational workloads by combining the multiple distributed Resource Managers an organisation is using, into a single aggregated view, allowing batch jobs to be directed to the best location for execution, using local resource managers such as LSF, PBSPro, SGE, LoadLeveler. SynfiniWay is able to schedule and execute services which are deployed on a mixed interlinked set of local resource managers.

Chapter-7

Yaim and Remote Procedure Call

Yaim

YAIM's official logo - The Yak

YAIM - YAIM Ain't an Installation Manager, is a tool to configure the middleware of the EGEE project namely the LCG and gLite software packages.

Description

Yaim is a software to configure Grid services, but YAIM also can be used as a general purpose configuration tool completely independent from the Grid middleware. The aim of YAIM is to provide simple configuration methods that can be used to set up uniform Grid sites but can also be easily adapted to meet the needs of larger sites. To ensure that all system administrators can adapt YAIM to the local requirements, it has been implemented as a set of bash scripts. To support EGEE's component based release model YAIM 4 has been modularized and a YAIM core is supplemented by component specific scripts, distributed as separate rpms.

Structure

YAIM's modular structure allows a distributed and asynchronous development which is essential in implementing the quickly changing configuration requirements of the Grid Middleware.

The hierarchical configuration storage - which is to reflect the architecture of a Grid Site - and the configurable function behavior make it easy to implement the local settings along with YAIM's default configuration.

Multi-level logging messages could provide detailed information on the ongoing configuration process.

YAIM's directory structure

When a YAIM 4 module is installed the following directory structure is created under /opt/glite/yaim:

- **/functions/**: Contains the functions the configure each node types. They are all bash scripts.
- **/functions/local/**: Site administrators can put here their own function definition files. Those will overwrite the default ones coming with the YAIM rpms. The file name has to be the same as the function name defined inside it.
- **/functions/pre/**: Function definition files having the same name as the original YAIM function but the defined function has the '_pre' suffix. This function - if exists - will be executed before the main function.
- **/functions/post/**: Function definition files having the same name as the original YAIM function but the defined function has the '_post' suffix. This function - if exists - will be executed after the main function.
- **/node-info.d/**: Contains a set of files coming with different YAIM modules and they contain the list of functions to be executed during the configuration the that given nodeype. Their name should be the lower-case variant of the node type.
- **/defaults/**:The filenames in this directory are having the same format as they have in node-info.d with a .pre and/or .post suffix. They are sourced correspondingly before and after the main site-info.def and their purpose is to give meaningful default - if possible - values to the variables used by the given module.
- **/bin/**: Contains the main yaim executable.
- **/log/**: The location of YAIM's logfile, yaimlog.
- **/examples/**: This directory contains an example configuration storage. Its structure is explained in the next section.

YAIM's configuration storage

The /examples/ directory in /opt/glite/yaim is just a guideline and serious site admins should not store their configuration files in that location. YAIM allows having the site's configuration in a well separated and protected place. The configuration files of a site are to be stored in a directory structure having a fixed layout relative to the site's main site-info.def file. The site-info.def file has to sit in a directory (ex.: /root/siteinfodir/) which could contain the followings:

- **/site-info.def**: This is the main configuration file of the site, to be sourced first and other optional configuration files will overwrite the values defined here.

- **/services/**: Site administrators can place files into this directory with their names having the same format as in /opt/glite/yaim/defaults. These files are sourced only

when the given service is configured, thus enabling service-specific configuration settings.

- **/nodes/**: Each file in this directory should be the fully qualified name of a host. Only when that host is to be configured, the corresponding file will be sourced which allows host-specific configuration steps to be performed.
- **/vo.d/**: The entries in this directory are to be named as the lower-case version of a VO to be supported. The file can contain VO specific variables and will overwrite the ones defined in the main site-info.def file.
- **/users.conf/**, **groups.conf**: Usually they reside on the same level in the directory hierarchy where the main site-info.def does, but their location is configurable via the USERS_CONF and GROUPS_CONF variables.

The bin/yaim executable

Usage: `/opt/glite/yaim/bin/yaim <action> <parameters>`

Actions:

```
-i | --install     : Install one or several meta package.
                     Compulsory parameters: -s, -m

-c | --configure   : Configure already installed services.
                     Compulsory parameters: -s, -n

-r | --runfunction : Execute a configuration function.
                     Compulsory parameters: -s, -f
                     Optional parameters  : -n

-v | --verify      : Goes through on all the functions and checks that
                     the necessary variables required for a given
                     configuration target are all defined in site-
info.def.
                     Compulsory parameters: -s -n

-d | --debug       : Turns "set -x" on, rude debug information
                     for development. Probably you don't want to use
it.

-e | --explain     : Doesn't perform configuration but explains what
the
                     functions are doing by printing out the comments
                     found inside them.
                     Compulsory parameters: -s -n

-h | --help        : Prints out this help.
```

Parameters:

```
-s | --siteinfo:   : Location of the site-info.def file
-m | --metapackage : Name of the metapackage(s) to install
-n | --nodetype    : Name of the node type(s) to configure
-f | --function    : Name of the functions(s) to execute
```

Examples:

```
Installation:
  /opt/glite/yaim/bin/yaim -i -s /root/siteinfo/site-info.def -m glite-
SE_dpm_mysql

Configuration:
  /opt/glite/yaim/bin/yaim -c -s /root/siteinfo/site-info.def -n
SE_dpm_mysql

Running a function:
  /opt/glite/yaim/bin/yaim -r -s /root/siteinfo/site-info.def -n
SE_dpm_mysql -f config_mkgridmap

Verify your site-info.def:
  /opt/glite/yaim/bin/yaim -v -s /root/siteinfo/site-info.def -n
SE_dpm_mysql
```

The configuration flow

When launching the configuration the different configuration files will be sourced and will overwrite each other in the following order (supposing that we store the site configuration in /root/siteinfo/ directory and configuring service myservice on host myhost and supporting VO myvo):

1. /opt/glite/yaim/defaults/site-info.pre
2. /opt/glite/yaim/defaults/myservice.pre
3. /root/siteinfo/site-info.def
4. /opt/glite/yaim/defaults/site-info.post
5. /opt/glite/yaim/defaults/myservice.post
6. /root/siteinfo/nodes/mynode
7. /root/siteinfo/vo.d/myvo
8. /opt/glite/yaim/node-info.d/myservice

Then for each function myfunc defined in /opt/glite/yaim/node-info.d/myservice it executes the followings if they are defined:

1. /opt/glite/yaim/functions/pre/myfunc.pre
2. /opt/glite/yaim/functions/myfunc or /opt/glite/yaim/functions/local/myfunc
3. /opt/glite/yaim/functions/post/myfunc

YAIM's logo - The Yak

The Yak (Bos grunniens) is a long-haired humped domestic bovine found in Tibet and throughout the Himalayan region of south Central Asia, as well as in Mongolia. The yak is not a gnu.

The author of the logo is **David O'Callaghan**. When he was asked why he has chosen the yak to be the logo of YAIM, he said:

"*The Jargon File defines 'yak shaving' as "Any seemingly pointless activity which is actually necessary to solve a problem which solves a problem which, several levels of recursion later, solves the real problem you're working on. I think this describes the process of installing and configuring grid systems quite well!*

YAIM aims to take the pain out of grid system administration. So, I thought a yak would make a good logo for YAIM as we no longer have to spend so much time shaving the yak!"

Remote procedure call

In computer science, a **remote procedure call** (**RPC**) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called **remote invocation** or **remote method invocation**.

Note that there are many different (often incompatible) technologies commonly used to accomplish this.

History and origins

The idea of RPC (Remote Procedure Call) goes back at least as far as 1976, when it was described in RFC 707. One of the first business uses of RPC was by Xerox under the name "Courier" in 1981. The first popular implementation of RPC on Unix was Sun's RPC (now called ONC RPC), used as the basis for NFS (Sun).

Another early Unix implementation was Apollo Computer's Network Computing System (NCS). NCS later was used as the foundation of DCE/RPC in the OSF's Distributed Computing Environment (DCE). A decade later Microsoft adopted DCE/RPC as the basis of the Microsoft RPC (MSRPC) mechanism, and implemented DCOM on top of it. Around the same time (mid-90's), Xerox PARC's ILU, and the Object Management Group's CORBA, offered another RPC paradigm based on distributed objects with an inheritance mechanism.

Message passing

An RPC is initiated by the *client*, which sends a request message to a known remote *server* to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process. There are many variations and subtleties in various implementations, resulting in a variety of different

(incompatible) RPC protocols. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution).

An important difference between remote procedure calls and local calls is that remote calls can fail because of unpredictable network problems. Also, callers generally must deal with such failures without knowing whether the remote procedure was actually invoked. Idempotent procedures (those that have no additional effects if called more than once) are easily handled, but enough difficulties remain that code to call remote procedures is often confined to carefully written low-level subsystems.

Sequence of events during a RPC

1. The client calls the Client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
3. The kernel sends the message from the client machine to the server machine.
4. The kernel passes the incoming packets to the server stub.
5. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

Standard contact mechanisms

To let different clients access servers, a number of standardized RPC systems have been created. Most of these use an interface description language (IDL) to let various platforms call the RPC. The IDL files can then be used to generate code to interface between the client and server. The most common tool used for this is RPCGEN.

Other RPC analogues

RPC analogues found elsewhere:

- Java's Java Remote Method Invocation (Java RMI) API provides similar functionality to standard UNIX RPC methods.
- Modula-3's Network Objects, which were the basis for Java's RMI
- XML-RPC is an RPC protocol that uses XML to encode its calls and HTTP as a transport mechanism.
- JSON-RPC is an RPC protocol that uses JSON encoded messages
- Microsoft .NET Remoting offers RPC facilities for distributed systems implemented on the Windows platform.
- RPyC implements RPC mechanisms in Python, with support for asynchronous calls.
- Pyro Object Oriented form of RPC for Python.
- Etch (protocol) framework for building network services.
- Facebook's Thrift protocol and framework.

- CORBA provides remote procedure invocation through an intermediate layer called the "Object Request Broker"
- DRb allows Ruby programs to communicate with each other on the same machine or over a network. DRb uses remote method invocation (RMI) to pass commands and data between processes.
- AMF allows Flex applications to communicate with back-ends or other applications that support AMF.
- Libevent provides a framework for creating RPC servers and clients.
- Windows Communication Foundation is an application programming interface in the .NET Framework for building connected, service-oriented applications.

Chapter-8

OpenLink ODBC Drivers and Oracle Fusion Middleware

OpenLink ODBC Drivers

High-performance, standards-compliant data access drivers, the **OpenLink Drivers for Open Database Connectivity (ODBC)** were arguably the first ODBC driver implementation available for non-Windows platforms such as Unix-like environments. OpenLink has continued to innovate in their drivers with the incorporation of a multi-tier security model, simplification of single-tier and multi-tier administration, and performance that often beats the DBMS-vendor's own drivers, while providing transparent, ODBC-based access to data sources (such as databases) from Desktop Productivity Tools, Application Development Environments, and Web & Internet Points of Presence.

History

First shipped in 1992, by the then PAL Consulting Ltd, the **OpenLink Drivers for Open Database Connectivity (ODBC)** started as Windows-only components, as ODBC was then a Windows-only technology, and were strictly "Single-Tier." At first, they only supported connections to Progress databases, but this quickly expanded to include Unify, Ingres, Oracle, Informix, Sybase, and Microsoft SQL Server databases.

Innovative from the start, in 1993, OpenLink Software introduced a new product that replaced the database-specific proprietary communication layers with its own network layer for serving ODBC clients. The new architecture used a Session Rulebook and a generic client driver, used for connections to all supported database engines, for Windows, Unix, Macintosh, and OS/2 clients. Server components were shipped to run on Windows NT, OS/2, VMS, and Unix-like systems. All database engines supported by the Single-Tier were also handled by the new Multi-Tier implementation.

A Philosophy of Open Access

OpenLink believed that the benefits of ODBC should be available to users on any platform. However, there was no ODBC Driver Manager available on any other platform. Therefore, initially, the Windows-only ODBC drivers were accompanied by drivers for UDBC, or Universal Database Connectivity, for use on Unix-like and other non-Windows platforms. In this packaging, all functions of the ODBC Driver Manager were built directly into the data access driver, getting around limitations of the Unix-like environments which generally could not handle dynamic libraries as they do today. This shifted the generic abstraction layer completely from the Driver Manager to the Driver, and was never intended as a permanent solution.

Inspired by their UDBC products, Ke Jin partnered with OpenLink to develop the non-Windows ODBC Driver Manager which they named iODBC, for Independent Open Database Connectivity. This project brought full-powered ODBC support to many non-Windows operating systems including Solaris, AIX, HP-UX, OpenVMS, the BSD and Linux variants, and Mac OS 9.

Open Access Doesn't Mean Unprotected

By splitting the elements of the driver in this way, OpenLink was able to incorporate several significant layers of centrally-administered additional security. The Session Rulebook restricts client access to the Database server based on multiple-access criteria including the requested DBMS Engine, Database Catalog and/or Schema, and the requesting client Username, Application, OS, and Hostname.

This gives the company Network/Database Administrator ultimate control of who or what groups of users are allowed access to the database, and what sort of access they get. Possible restrictions include limiting any Sales user's query result sets to 100 records (so the contents of an entire customer database cannot be taken to a competitor, and Cartesian product-based Denial-of-Service on the database server instance are impossible); restricting Microsoft Access users to Read Only connections (minimizing accidental database wipe-outs); entirely preventing connections from outside the enterprise LAN IP address space.

Wire-level encryption between the OpenLink client and server components was also added to the mix circa 1994 , long before SSL was an available option -- and SSL itself was also added in due course (circa 2004).

The security models and features provided by OpenLink's Multi-Tier Rulebook remain a central element of their Multi-Tier implementation, and have yet to be matched in any other data access implementation, ODBC or otherwise.

The Need for Speed

The X/Open and SQL/CLI API efforts were initially supported by the various DBMS vendors, at a time when they recognized that more client tools would be available for use with each DBMS if the application vendors could code to a single API, rather than refactoring their application for each and every supported DBMS.

When ODBC started getting significant uptake, the DBMS vendors seemed to forget this vision. Directly and indirectly, they began supporting grumblings from the field, that ODBC was slow by nature. This generalized knock on the protocol was disproven by published studies, but there was still a hesitance to trust performance claims by ODBC driver vendors.

OpenLink recognized that there were several ways to implement an ODBC driver, and further that their Multi-Tier "Enterprise Edition" was not the fastest of these, even with its Multi-Threaded architecture and support for Advanced Data Access API calls. Although the security and other administrative features of the Multi-Tier implementation have significant value to the enterprise, there are times that raw performance is more important in a deployment.

To deliver the required performance, OpenLink expanded and improved their Single-Tier "Lite Edition" offerings. These drivers typically require the additional installation of database-specific networking components (e.g., Oracle Instant Client, Progress Client Access) on the client host. In some cases, where the database and/or its network protocol is open source (e.g., PostgreSQL, MySQL, TDS for Sybase & Microsoft SQL Server), it is built directly into the driver, and no additional component installation is required.

Once installed, the Lite Edition provides connectivity to both local and remote databases. The user must generally specify a few connection attributes, such as the database instance listening port and hostname, in addition to the instance name. Of course, these Single-Tier drivers maintained the multi-threaded architecture and inherited full support for Advanced Data Access API calls that had been implemented in the Enterprise Edition.

Testing with the open source OpenLink ODBC Bench in one's own environment shows that the Lite Edition drivers measure up to -- and often surpass -- the performance of all other drivers in their class, including those from the DBMS vendor. The Enterprise Edition can also be seen to be no slouch.

Simplify, Simplify

More recently, complaints about ODBC have focused on the perceived complexity of setting up connections, especially from the end-user's perspective. OpenLink innovation has continued to match.

First, Zeroconf (also known as Bonjour) functionality was implemented in the existing components.

One of the original features of the Enterprise Edition was the ability to force database connection attributes on the user, but the user had to know the basic connection attributes of the Multi-Tier components. With Zeroconf implementation, even this was no longer required, as the administrator could now register "network DSNs" similarly to the old practice of registering network printers -- and the user could choose them through their local driver manager's setup dialogs, just as they already did with their Printer setup dialogs.

Some DBMS vendors added similar network broadcast or advertising features features to their database engines, and the Lite Edition drivers were enhanced to recognize such broadcasts and make those instances available to users.

Finally, a new breed of driver, the "Express Edition" was created, enabling connections in most cases when the user knows only the instance name. More importantly, the Express Edition requires no secondary component installation, as it wraps an ODBC driver around a Java-based wire protocol library, typically from the DBMS vendor.

Universal Benefits of Universal Data Access

Through any of these drivers, OpenLink provides the freedom to mix and match "best of class" IT infrastructure components, preventing vendor lock-in that can impede enterprise agility.

By supporting open standards and specifications, these components help preserve existing investment in IT infrastructure, while empowering Information and Knowledge Workers to create new market opportunities.

- Support Advanced Data Access API functionality
- Support all ODBC Scrollable Cursor Models
- Enhance the security of as-shipped DBMS Engines (Multi-Tier only)
- Deliver performance benefits over DBMS vendor-supplied drivers (primarily Single-Tier)

Oracle Fusion Middleware

Oracle Fusion Middleware (OFM, also known as **Fusion Middleware**) consists of several software products from Oracle Corporation. OFM spans multiple services, including Java EE and developer tools, integration services, business intelligence, collaboration, and content management. OFM depends on open standards such as BPEL, SOAP, XML and JMS.

Oracle Fusion Middleware provides software for the development, deployment, and management of service-oriented architecture (SOA). It includes what Oracle calls "hot-

pluggable" architecture, designed to facilitate integration with existing applications and systems from other software vendors such as IBM, Microsoft, and SAP AG.

Evolution

Many of the products included under the OFM banner do not themselves qualify as middleware products: "Fusion Middleware" essentially represents a re-branding of many of Oracle products outside of Oracle's core database and applications-software offerings—compare Oracle Fusion.

According to Oracle, by 2006 over 30,000 organizations had become Fusion Middleware customers, including over 35 of the world's 50 largest companies and more than 750 of the *BusinessWeek* Global 1000, with OFM also supported by 7,500 partners.

In order to provide standards-based software to assist with business process automation, HP has incorporated OFM into its "service-oriented architecture (SOA) portfolio".

Oracle leveraged its Configurable Network Computing (CNC) technology acquired from its PeopleSoft/JD Edwards 2005 purchase.

Oracle Fusion Applications, based on Oracle Fusion Middleware, were finally released in September, 2010.

Assessments

In January 2008 Oracle Universal Content Management won *InfoWorld*'s "Technology of the Year" award for "Best Enterprise Content Manager", with Oracle SOA Suite winning the award for "Best Enterprise Service Bus".

In 2007 Gartner, Inc. wrote that "OFM has reached a degree of completeness that puts it on par with, and in some cases ahead of, competing software stacks", and reported revenue from the suite of over US$1 billion during FY06, estimating the revenue from the genuinely middleware aspects at US$740 million.

Oracle Fusion Middleware components

- Enterprise application server
  - Oracle Weblogic Server
  - Oracle Application Server
  - JRockit (a JVM)
  - Tuxedo (software)
- Integration- and process-management
  - BPEL Process Manager
  - Business activity monitoring
  - business rules
  - Business Process Analysis Suite

    - Business process management
    - Oracle Data Integrator (ODI): an application using the database for set-based data integration
    - Enterprise connectivity (adapters)
    - Oracle Enterprise Messaging Service
    - Oracle Enterprise Service Bus
    - Oracle Application server B2B
    - Oracle Service Registry
    - Oracle Web Services Manager (OWSM), a security and monitoring product for web services
- Application development tools
    - Oracle Application Development Framework
    - JDeveloper
    - Oracle SOA Suite
    - TopLink, a Java object-relational mapping package
    - Oracle Forms services
    - Oracle Developer Suite
- Business intelligence
    - Oracle Business Intelligence 10g
    - Oracle Business Activity Monitoring (Oracle BAM)
    - Oracle Crystal Ball - does "predictive modeling, forecasting, and simulation"
    - Oracle Discoverer
    - Data hubs
    - Oracle BI Publisher
    - Oracle Reports services
- Systems management
    - Oracle Enterprise Manager
    - Web services manager
- User interaction
    - Oracle Beehive collaboration software
    - Oracle Portal
    - Oracle WebCenter
    - Real-time collaboration
    - Unified messaging
    - Workspaces
- Content management
    - Oracle Imaging and Process Management
    - Web content management
    - Records management
    - Enterprise search
    - Digital asset management
    - Email archiving
    - Oracle Universal Content Management: In November 2006 Oracle Corporation acquired Stellent, a software-development company (based in Eden Prairie, Minnesota) which provided content management systems.

Stellent's primary product, "Universal Content Management" (UCM), the foundation of most of its other content-management products, became *Oracle Universal Content Management* as a part of the Oracle Fusion Middleware stack. Oracle retained the name "Stellent" for this suite of applications. (Before 2001 Stellent had used the name "Intranet Solutions" and called its product first "IntraDoc!", then briefly "Xpedio".

- Identity management
  - Enterprise Single sign-on
  - Oracle Entitlements Server
  - Oracle Identity Manager
  - Oracle Access Manager
  - Oracle Adaptive Access Manager
  - Oracle Information Rights Management
- Grid infrastructure
  - Services registry
  - application-server security

Integration, pricing and bundling

Apart from selling licenses to run OFM components, Oracle Corporation also markets a managed option via the SaaS Oracle On Demand service.

Chapter-9

ProActive

	ProActive
Developer(s)	OW2 Consortium
Written in	Java
Operating system	Cross-platform
Type	Grid Computing
License	GNU General Public License

ProActive is Java grid middleware for parallel, distributed, and multi-threaded computing. It is developed by the OW2 Consortium, including INRIA, CNRS, University of Nice Sophia Antipolis, and ActiveEon. It is open-source software released under the GPL license.

ProActive provides a comprehensive framework and parallel programming model to simplify the programming and execution of parallel applications running on multi-core processors, distributed on Local Area Network (LAN), on clusters and data centers, on intranets, and on Internet grids.

The ProActive programming model combines the active object design pattern with futures objects.

Programming model

The model was created by Denis Caromel, professor at University of Nice Sophia Antipolis. Several extensions of the model were made later on by members of the OASIS team at INRIA. The book *A Theory of Distributed Objects* presents the ASP calculus that formalizes ProActive features, and provides formal semantics to the calculus, together with properties of ProActive program execution.

Active objects

Active objects are the basic units of activity and distribution used for building concurrent applications using ProActive. An active object runs with its own thread. This thread only executes the methods invoked on this active object by other active objects, and those of the passive objects of the subsystem that belongs to this active object. With ProActive, the programmer does not have to explicitly manipulate Thread objects, unlike in standard Java.

Active objects can be created on any of the hosts involved in the computation. Once an active object is created, its activity (the fact that it runs with its own thread) and its location (local or remote) are perfectly transparent. Any active object can be manipulated as if it were a passive instance of the same class.

An *active object* is composed of two objects: a *body*, and a standard Java object. The body is not visible from the outside of the active object.

The body is responsible for receiving calls (or *requests*) on the active object and storing them in a queue of pending calls. It executes these calls in an order specified by a synchronization policy. If a synchronization policy is not specified, calls are managed in a "First in, first out" (FIFO) manner.

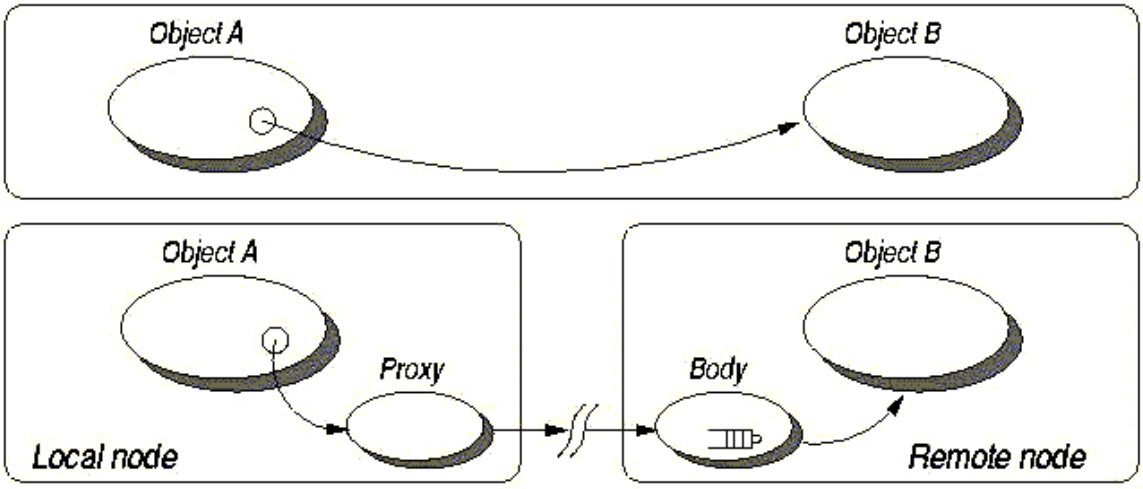
The thread of an active object then chooses a method in the queue of pending requests and executes it. No parallelism is provided inside an active object; this is an important decision in ProActive's design, enabling the use of "pre-post" conditions and class invariants.

On the side of the subsystem that sends a call to an active object, the active object is represented by a *proxy*. The proxy generates future objects for representing future values, transforms calls into Request objects (in terms of metaobject, this is a reification) and performs deep copies of passive objects passed as parameters.

Active object basis

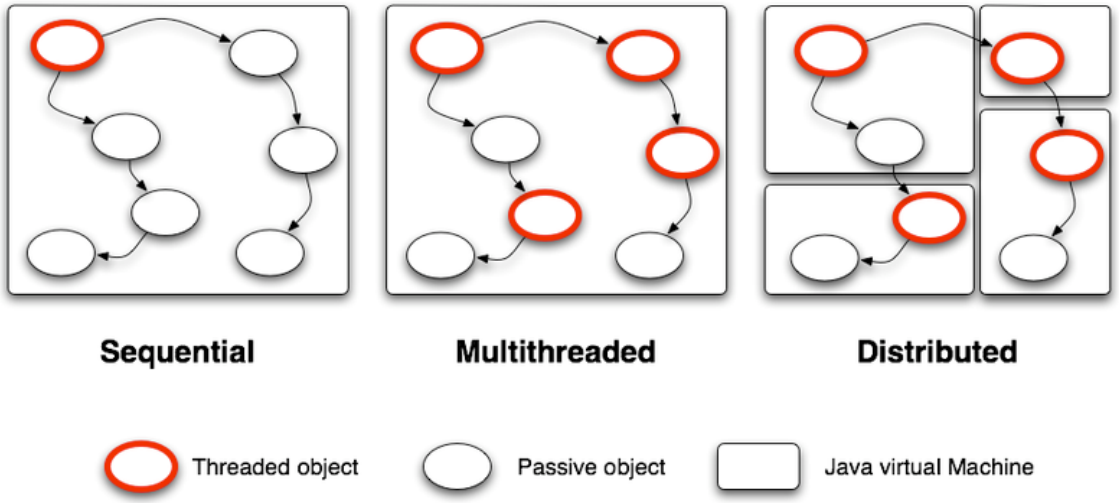
ProActive is a library designed for developing applications in the model introduced by Eiffel//, a parallel extension of the Eiffel programming language.

In this model, the application is structured in *subsystems*. There is one active object (and therefore one thread) for each subsystem, and one subsystem for each active object (or thread). Each subsystem is thus composed of one active object and any number of passive objects—possibly no passive objects. The thread of one subsystem only executes methods in the objects of this subsystem. There are no "shared passive objects" between subsystems.



A call onto an active object, as opposed to a call onto passive one

These features impact the application's topology. Of all the objects that make up a subsystem—the active object and the passive objects—only the active object is known to objects outside of the subsystem. All objects, both active and passive, may have references onto active objects. If an object *o1* has a reference onto a passive object *o2*, then *o1* and *o2* are part of the same subsystem.



The model: Sequential, multithreaded, distributed

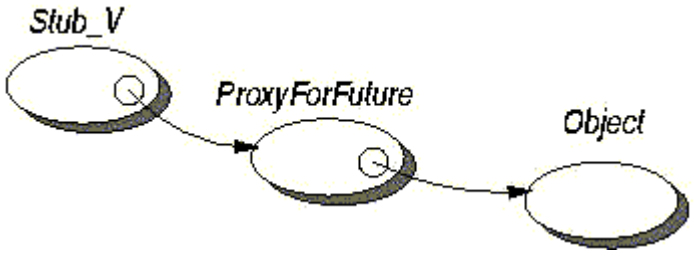
This has also consequences on the semantics of message-passing between subsystems. When an object in a subsystem calls a method on an active object, the parameters of the call may be references on passive objects of the subsystem, which would lead to shared passive objects. This is why passive objects passed as parameters of calls on active

objects are always passed by deep-copy. Active objects, on the other hand, are always passed by reference. Symmetrically, this also applies to objects returned from methods called on active objects.

Thanks to the concepts of asynchronous calls, futures, and no data sharing, an application written with ProActive doesn't need any structural change—actually, hardly any change at all—whether it runs in a sequential, multi-threaded, or distributed environment.

Asynchronous calls and futures

Whenever possible, a method call on an active object is reified as an asynchronous request. If not possible, the call is synchronous, and blocks until the reply is received. If the request is asynchronous, it immediately returns a *future object*.



A future object

The future object acts as a placeholder for the result of the not-yet-performed method invocation. As a consequence, the calling thread can go on with executing its code, as long as it doesn't need to invoke methods on the returned object. If the need arises, the calling thread is automatically blocked if the result of the method invocation is not yet available. Although a future object has structure similar to that of an active object, a future object is not active. It only has a Stub and a Proxy.

A simple example

The code excerpt below highlights the notion of future objects. Suppose a user calls a method `foo` and a method `bar` from an active object `a`; the `foo` method returns void and the `bar` method returns an object of class `V`:

```
// a one way typed asynchronous communication towards the (remote) AO a
// a request is sent to a
a.foo (param);

// a typed asynchronous communication with result.
// v is first an awaited Future, to be transparently filled up after
// service of the request, and reply
V v = a.bar (param);
...
// use of the result of an asynchronous call.
// if v is still an awaited future, it triggers an automatic
```

```
// wait: Wait-by-necessity
v.gee (param);
```

When `foo` is called on an active object `a`, it returns immediately (as the current thread cannot execute methods in the other subsystem). Similarly, when `bar` is called on `a`, it returns immediately but the result `v` can't be computed yet. A future object, which is a placeholder for the result of the method invocation, is returned. From the point of view of the caller subsystem, there is no difference between the future object and the object that would have been returned if the same call had been issued onto a passive object.

After both methods have returned, the calling thread continues executing its code as if the call had been effectively performed. The role of the future mechanism is to block the caller thread when the `gee` method is called on `v` and the result has not yet been set : this inter-object synchronization policy is known as *wait-by-necessity*.

Chapter-10

Jsonwsp

JSON-WSP is short for JavaScript Object Notation Web-Service Protocol. Simply put it is a web-service protocol that uses JSON for service description, requests and responses. It is very much inspired from JSON-RPC, but The lack of a service description specification with documentation in JSON-RPC sparked the design of JSON-WSP.

The description format has the same purpose for JSON-WSP as WSDL has for SOAP or IDL for CORBA which is to describe the types and methods used in a given service. It also describes intertype relations (i.e. nestet types) and defines which types are expected as method arguments and which types the user can expect to receive as method return values. Finally the description opens the possibility to add documentation on service, method, parameter and return levels.

Communication between clients and a JSON-WSP server is carried out using HTTP POST requests and responses, with the JSON objects as data with the content-type application/json.

Specifications

JSON-WSP consists of 4 JSON object specifications:

Specification	Description
description	Service description specification (like WSDL). This specification describes methods, method parameters, types and return types. It also supports user documentation on service, method and parameter levels.
request	Specification for JSON requests. It contains information about which method that is to be invoked and all the arguments for the method call. Arguments in the request must obey the parameter definition of the same method described in the corresponding

	JSON-WSP description.
response	Specification for JSON responses. The response object contains the reslult of a service method invokation. The return type must obey the defined return type of the same method in the corrosponding JSON-WSP description.
fault	Specification for JSON fault responses. The fault object contains a fault code and a fault string. The fault information specifies whether the fault occurred on the client or server side. Depending on the server side service framework more detailed information can be extracted, i.e. the filename and line number where the fault occurred.

Understanding the specification notation

Building blocks

- If the name of the building-block being defined starts with **rx-** it means that the definition is a regular expression. In these definitions square brackets have the role of defining character classes and parentheses have the role of defining capturing groups.
- In all other cases square brackets notate lists and parentheses notate either a decition:
- `( d1 | d2 | ... )`

  a repetition of 0-many:

  `( ... )*`

  a repetition 1-many:

  `( --- )+`

  or something optional:

  `( ... )?`

Common building-blocks

```
<rx-freetext> = ".*"

<rx-identifier> = "[a-zA-Z_][a-zA-Z0-9_]*"

<rx-number> = [0-9]+

<rx-boolean> = (true|false)

<key> = <rx-identifier>
```

```
<primitive-value> = ( <rx-freetext> | <rx-number> | <rx-boolean> )

<value> = (
    <primitive-value> |
    [ ( <value>, )* ] |
    { ( <key>: <value>, )* } )

<method-name> = <rx-identifier>

<service-name> = <rx-identifier>
```

The JSON-WSP description object

Additional Building-blocks

```
<primitive> = ( "string" | "number" | "float" )

<service-locator> = <rfc-1738 compliant string>

<type-name> = <rx-identifier>

<member-name> = <rx-identifier>

<multi-type> = ( <primitive> | <type-name> | [<primitive>] | [<type-
name>] )

<doc-string> = <rx-freetext>

<param-name> = <rx-identifier>

<def-order> = <rx-number>

<param-optional> = <rx-boolean>
```

Specification

```
{
    "type": "jsonwsp/description",
    "version": "1.0",
    "servicename": <service-name>,
    "url": <service-locator>,
    "types": { (
        <type-name>: { (
            <member-name>: <multi-type> )+
        } )*
    },
    "methods": { (
        <method-name>: {
            "doc_lines": [ ( <doc-string>, )* ],
            "params": { (
                <param-name>: {
                    "doc_lines": [ ( <doc-string>, )* ],
                    "def_order": <def-order>,
                    "type": <multi-type>,
                    "optional": <param-optional>
                }, )*
```

```
            },
            "ret_info": {
                "doc_lines": [ ( <doc-string>, )* ],
                "type": <multi-type>
            }
        } )+
    }
}
```

Descriptions

<service-locator>: The service endpoint URL that accepts JSON-WSP POST request objects.

<service-name>: Service name is case sensitive. It identifies a specific service exposed on a specific server.

doc_lines: Each doc-string contained in a doc_lines list reflects a single line of documentation that relates to the parent object of the doc_lines.

The JSON-WSP request object

The request object contains information about which method to invoke and what arguments to invoke the method with. It also stores information about the type and version of itself.

The optional **mirror** value can be used to send information from the client which will then be reflected by the server and returned unchanged in the response object's **reflection** value. This feature allows clients to send multiple requests to a method and send request identification values that can be intercepted by the client's response handler. This is often nessecary from javascript if more than one request is being processed simultaneously by the server and the response order is unknown by the client.

Specification

```
{
    "type": "jsonwsp/request",
    "version": "1.0",
    "methodname": <method-name>,
    "args": { ( <key>: <value>, )* }(,
    "mirror": <value> )?
}
```

The JSON-WSP response object

Specification

The **reflection** value is an unchanged server reflection of the request object's **mirror** value. It is marked as optional because it is the client that controls via the request whether it is there or not.

```
{
    "type": "jsonwsp/response",
    "version": "1.0",
    "servicename": <service-name>,
    "methodname": <method-name>,
    "result": <value>(,
    "reflection": <value> )?
}
```

The JSON-WSP fault response object

Additional Building-blocks

```
<fault-code> = ( "incompatible" | "client" | "server" )

<fault-string> = <rx-freetext>

<fault-filename> = <rx-freetext>

<fault-lineno> = <rx-number>
```

Specification

```
{
    "type": "jsonwsp/fault",
    "version": "1.0",
    "fault": {
        "code": <fault-code>,
        "string": <fault-string>,
        ("detail": [ ( <fault-string>, )* ] ,)?
        ("filename": <fault-filename>,)?
        ("lineno": <fault-lineno>,)?
    },
}
```

Descriptions

<fault-code>: The meanings of the possible fault-codes:

- "incompatible": Client version of JSON-WSP is incompatible with the server version of JSON-WSP. Typically you will encounter this type of fault-code if there is a version major in difference between the client and the server.
- "server": An error occurred on the server side after the client request has been successfully consumed.
- "client": The clients request could not be consumed by the server due to incorrect format or missing required arguments etc.

Real world example

Description

```
{
    "type": "jsonwsp/description",
    "version": "1.0",
```

```
    "servicename": "UserService",
    "url":
"http://testladon.org:80/proxy.php?path=UserService/jsonwsp",
    "types": {
        "Group": {
            "group_id": "number",
            "display_name": "string",
            "name": "string",
            "members": ["User"]
        },
        "User": {
            "username": "string",
            "user_id": "number",
            "mobile": "string",
            "age": "number",
            "given_name": "string",
            "surname": "string"
        },
        "CreateUserResponse": {
            "user_id": "number",
            "success": "boolean"
        }
    },
    "methods": {
        "listUsers": {
            "doc_lines": ["List Users that have a username, given_name
or surname that matches a given filter."],
            "params": {
                "name_filter": {
                    "def_order": 1,
                    "doc_lines": ["String used for filtering the
resulting list of users."],
                    "type": "string",
                    "optional": false
                }
            },
            "ret_info": {
                "doc_lines": ["List of users."],
                "type": ["User"]
            }
        },
        "listGroups": {
            "doc_lines": ["List Groups that have a name or display_name
that matches a given filter."],
            "params": {
                "name_filter": {
                    "def_order": 1,
                    "doc_lines": ["String used for filtering the
resulting list of groups."],
                    "type": "string",
                    "optional": false
                }
            },
            "ret_info": {
                "doc_lines": ["List of groups."],
                "type": ["Group"]
            }
```

```
        },
        "createUser": {
            "doc_lines": ["Create a new user account."],
            "params": {
                "username": {
                    "def_order": 1,
                    "doc_lines": ["Unique username for the new user
account."],
                    "type": "string",
                    "optional": false
                },
                "given_name": {
                    "def_order": 2,
                    "doc_lines": ["First name."],
                    "type": "string",
                    "optional": false
                },
                "surname": {
                    "def_order": 3,
                    "doc_lines": ["Last name."],
                    "type": "string",
                    "optional": false
                },
                "mobile": {
                    "def_order": 4,
                    "doc_lines": ["Optional mobile number."],
                    "type": "string",
                    "optional": true
                },
                "age": {
                    "def_order": 5,
                    "doc_lines": ["Optional age of the person behind
the account."],
                    "type": "number",
                    "optional": true
                }
            },
            "ret_info": {
                "doc_lines": [],
                "type": "CreateUserResponse"
            }
        }
    }
}
```

Service call 1

Request

```
{
    "type": "jsonwsp/request",
    "version": "1.0",
    "methodname": "createUser",
    "args": {
        "username": "bettyw",
        "given_name": "Betty",
        "surname": "Wilson",
```

```
        "mobile": "555-3423444"
    },
    "mirror": {
        "id": 2
    }
}
```

Response

```
{
    "type": "jsonwsp/response",
    "version": "1.0",
    "servicename": "UserService",
    "method": "createUser",
    "result": {
        "user_id": 324,
        "success": true
    }
    "reflection": {
        "id": 2
    }
}
```

Service call 2

Request

```
{
    "type": "jsonwsp/request",
    "version": "1.0",
    "methodname": "listUsers",
    "args": {
        "name_filter": "jack"
    }
}
```

Response

```
{
    "type": "jsonwsp/response",
    "version": "1.0",
    "servicename": "UserService",
    "method": "listUsers",
    "result": [{
        "username": "jackp",
        "user_id": 153,
        "mobile": "555-377843",
        "age": 34,
        "given_name": "Jack",
        "surname": "Petersen"
    }, {
        "username": "bradj",
        "user_id": 321,
        "mobile": "555-437546",
        "age": 27,
        "given_name": "Brad",
        "surname": "Jackson"
```

```
        }]
    }
```

Chapter-11

SOAP and SOAPjr

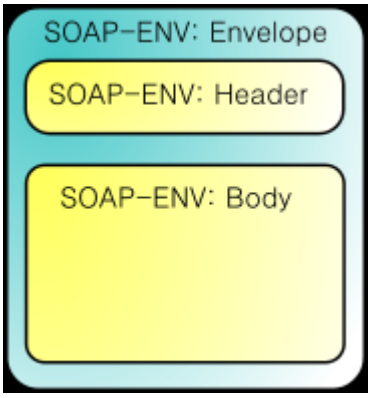
SOAP

SOAP, originally defined as **Simple Object Access Protocol**, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on Extensible Markup Language (XML) for its message format, and usually relies on other Application Layer protocols, most notably Remote Procedure Call (RPC) and Hypertext Transfer Protocol (HTTP), for message negotiation and transmission. SOAP can form the foundation layer of a web services protocol stack, providing a basic messaging framework upon which web services can be built. This XML based protocol consists of three parts: an envelope, which defines what is in the message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing procedure calls and responses.

As an example of how SOAP procedures can be used, a SOAP message could be sent to a web-service-enabled web site such as a real-estate price database, with the parameters needed for a search. The site would then return an XML-formatted document with the resulting data, e.g., prices, location, features. With the data being returned in a standardized machine-parseable format, it can then be integrated directly into a third-party web site or application.

The SOAP architecture consists of several layers of specifications: for message format, Message Exchange Patterns (MEP), underlying transport protocol bindings, message processing models, and protocol extensibility. SOAP is the successor of XML-RPC, though it borrows its transport and interaction neutrality and the envelope/header/body from elsewhere (probably from WDDX).

History



SOAP structure

SOAP once stood for 'Simple Object Access Protocol' but this acronym was dropped with Version 1.2 of the standard. Version 1.2 became a W3C recommendation on June 24, 2003. The acronym is sometimes confused with **SOA**, which stands for Service-oriented architecture, but the two are unrelated.

SOAP was originally designed by Dave Winer, Don Box, Bob Atkinson, and Mohsen Al-Ghosein in 1998 in a project for Microsoft (where Atkinson and Al-Ghosein were already working at the time), as an object-access protocol. The SOAP specification is currently maintained by the XML Protocol Working Group of the World Wide Web Consortium.

After SOAP was first introduced, it became the underlying layer of a more complex set of Web Services, based on Web Services Description Language (WSDL) and Universal Description Discovery and Integration (UDDI). These services, especially UDDI, have proved to be of far less interest, but an appreciation of them gives a more complete understanding of the expected role of SOAP compared to how web services have actually evolved.

Specification

The SOAP specification defines the messaging framework which consists of:

- The **SOAP processing model** defining the rules for processing a SOAP message
- The **SOAP extensibility model** defining the concepts of SOAP features and SOAP modules
- The **SOAP underlying protocol binding** framework describing the rules for defining a binding to an underlying protocol that can be used for exchanging SOAP messages between SOAP nodes
- The **SOAP message construct** defining the structure of a SOAP message

Processing model

The SOAP processing model describes a distributed processing model, its participants, the **SOAP nodes** and how a SOAP receiver processes a SOAP message. The following SOAP nodes are defined:

- **SOAP sender**

A SOAP node that transmits a SOAP message.

- **SOAP receiver**

A SOAP node that accepts a SOAP message.

- **SOAP message path**

The set of SOAP nodes through which a single SOAP message passes.

- **Initial SOAP sender (Originator)**

The SOAP sender that originates a SOAP message at the starting point of a SOAP message path.

- **SOAP intermediary**

A SOAP intermediary is both a SOAP receiver and a SOAP sender and is targetable from within a SOAP message. It processes the SOAP header blocks targeted at it and acts to forward a SOAP message towards an ultimate SOAP receiver.

- **Ultimate SOAP receiver**

The SOAP receiver that is a final destination of a SOAP message. It is responsible for processing the contents of the SOAP body and any SOAP header blocks targeted at it. In some circumstances, a SOAP message might not reach an ultimate SOAP receiver, for example because of a problem at a SOAP intermediary. An ultimate SOAP receiver cannot also be a SOAP intermediary for the same SOAP message.

Transport methods

Both SMTP and HTTP are valid application layer protocols used as Transport for SOAP, but HTTP has gained wider acceptance as it works well with today's Internet infrastructure; specifically, HTTP works well with network firewalls. SOAP may also be used over HTTPS (which is the same protocol as HTTP at the application level, but uses an encrypted transport protocol underneath) with either simple or mutual authentication; this is the advocated WS-I method to provide web service security as stated in the WS-I Basic Profile 1.1.

This is a major advantage over other distributed protocols like GIOP/IIOP or DCOM which are normally filtered by firewalls. *SOAP over AMQP* is yet another possibility that some implementations support.

Message format

XML was chosen as the standard message format because of its widespread use by major corporations and open source development efforts. Additionally, a wide variety of freely available tools significantly eases the transition to a SOAP-based implementation. The somewhat lengthy syntax of XML can be both a benefit and a drawback. While it promotes readability for humans, facilitates error detection, and avoids interoperability problems such as byte-order (Endianness), it can slow processing speed and can be cumbersome. For example, CORBA, GIOP, ICE, and DCOM use much shorter, binary message formats. On the other hand, hardware appliances are available to accelerate processing of XML messages. Binary XML is also being explored as a means for streamlining the throughput requirements of XML.

Technical critique

Advantages

- SOAP is versatile enough to allow for the use of different transport protocols. The standard stacks use HTTP as a transport protocol, but other protocols such as JMS and SMTP are also usable.
- Since the SOAP model tunnels fine in the HTTP get/response model, it can tunnel easily over existing firewalls and proxies, without modifications to the SOAP protocol, and can use the existing infrastructure.

Disadvantages

- Because of the verbose XML format, SOAP can be considerably slower than competing middleware technologies such as CORBA. This may not be an issue when only small messages are sent. To improve performance for the special case of XML with embedded binary objects, the Message Transmission Optimization Mechanism was introduced.
- When relying on HTTP as a transport protocol and not using WS-Addressing or an ESB, the roles of the interacting parties are fixed. Only one party (the client) can use the services of the other. Developers must use polling instead of notification in these common cases.

SOAPjr

SOAPjr is a protocol specification for exchanging structured information in the implementation of Web services in computer networks. It is a hybrid of SOAP and JSON-RPC (abbreviated as "jr" in this case).

Introduction

SOAPjr is designed to create clean, fast, AJAX-style APIs and is analogous to the introduction of out of band signalling in the telephony world

Traditional SOAP is no longer the Simple Object Access Protocol it was initially designed to be. It can be bloated and overly verbose making it bandwidth hungry and slow. It is also based on XML, making it expensive to parse and manipulate - especially on mobile or embedded clients. However, its core envelope/head/body design pattern is useful for AJAX style APIs.

SOAPjr uses a similar Envelope/Head/Body model, using lightweight and easier to manipulate JSON.

In contrast to SOAP, JSON-RPC is overly simplistic and basically tunnels HTTP GET-style key/value pairs within a query string using JSON. However, within JSON-RPC there is no head/body separation, leaving metadata to pollute the main data space.

SOAPjr combines the best of these two concepts and is designed to create modern AJAX APIs that can easily be used by mobile devices, embedded systems or desktop browsers.

SOAPjr is an Open Source project with software released under the GPL and content under Creative Commons.

JSON-Schema definitions

The following SOAPjr entities are defined as JSON-Schemas.

- SOAPjr_basic_object
- SOAPjr_error_record
- SOAPjr_errors_object
- SOAPjr_request
- SOAPjr_response

The latest versions can also be downloaded in a single file.

Common data models

SOAPjr.org also aims to contribute to the creation of a common set of DMDs (Data Model Definitions) that may align with the JSON-schema proposal and Service Mapping Description Proposal so applications within specific domains can easily share data. The primary extension that SOAPjr may provide here is the use of consistent or standardised error codes.

Other resources that may inform this development are common data models utilised within microformats and RDF

Chapter-12

JSON-RPC and Java Remote Method Invocation

JSON-RPC

JSON-RPC is a remote procedure call protocol encoded in JSON. It is a very simple protocol (and very similar to XML-RPC), defining only a handful of data types and commands. In contrast to XML-RPC or SOAP, it allows for bidirectional communication between the service and the client, treating each more like peers and allowing peers to call one another or send notifications to one another. It also allows multiple calls to be sent to a peer which may be answered out of order.

History

Version	Description	Dated
1.0	Original version Currently considered *official* according to	2005
1.1 WD	Working draft Adds named parameters, adds specific error codes, and adds introspection functions.	2006-08-07
1.1 Alt	Suggestion for a simple JSON-RPC 1.1 Alternative proposal to 1.1 WD.	2007-05-06
1.2	Proposal A later revision of this document was renamed to 2.0.	2007-12-27
2.0	Specification proposal	2009-05-24
2.0 (Revised)	Specification proposal	2010-03-26

Usage

JSON-RPC works by sending a request to a server implementing this protocol. The client in that case is typically software wanting to call a single method of a remote system.

Multiple input parameters can be passed to the remote method as an array or object, whereas the method itself can return multiple output data as well. (This depends on the implemented version.)
A remote method is invoked by sending a request to a remote service using HTTP or a TCP/IP socket (starting with version 2.0). When Using HTTP, the content-type may be defined as `application/json`.

All transfer types are single objects, serialized using JSON. A request is a call to a specific method provided by a remote system. It must contain three certain properties:

- `method` - A String with the name of the method to be invoked.
- `params` - An Array of objects to be passed as parameters to the defined method.
- `id` - A value of any type, which is used to match the response with the request that it is replying to.

The receiver of the request must reply with a valid response to all received requests. A response must contain the properties mentioned below.

- `result` - The data returned by the invoked method. This must be null in case an error occurred invoking the method.
- `error` - A specified Error code if there was an error invoking the method, otherwise `null`.
- `id` - The id of the request it is responding to.

Since there are situations, no response is needed or not even desired, notifications were introduced. A notification is similar to a request except for the id, which is not needed because no response will be returned. In this case the `id` property should be omitted (Version 2.0) or be `null` (Version 1.0).

Examples

In these examples, `-->` denotes data sent to a service (*request*), while `<--` denotes data coming from a service. (Although this direction often is called *response* in client-server computing, depending on the JSON-RPC version it does not necessarily imply *answer to a request*).

Version 1.0

A simple request and response:

```
--> { "method": "echo", "params": ["Hello JSON-RPC"], "id": 1}
<-- { "result": "Hello JSON-RPC", "error": null, "id": 1}
```

This example shows parts of a communication from an example chat application. The chat service sends notifications for each chat message the client peer should receive. The

client peer sends requests to post messages to the chat and expects a positive reply to know the message has been posted.

```
...
--> {"method": "postMessage", "params": ["Hello all!"], "id": 99}
<-- {"result": 1, "error": null, "id": 99}
<-- {"method": "handleMessage", "params": ["user1", "we were just
talking"], "id": null}
<-- {"method": "handleMessage", "params": ["user3", "sorry, gotta go
now, ttyl"], "id": null}
--> {"method": "postMessage", "params": ["I have a question:"], "id":
101}
<-- {"method": "userLeft", "params": ["user3"], "id": null}
<-- {"result": 1, "error": null, "id": 101}
...
```

Because params field is an array of objects, the following format is also ok:

```
{
 "method": "methodnamehere",
 "params": [
            {
             "firstparam": "this contains information of the
firstparam.",
             "secondparam": 1121211234,
             "thirdparam": "this contains information of the
thirdparam."
            },
            {
             "fourthparam": "this is already a different object.",
             "secondparam": "there can be same name fields in different
objects.",
             "thirdparam": "this contains information of the
thirdparam."
            }
           ],
 "id": 1234
}
```

Version 1.1 (Working Draft)

The format of the contents of a request might be something like that shown below:

```
{
"version": "1.1",
"method": "confirmFruitPurchase",
"id": "194521489",
"params": [
    [ "apple", "orange", "pear" ],
    1.123
]
```

```
}
```

The format of a response might be something like this:

```
{
"version": "1.1",
"result": "done",
"error": null,
"id": "194521489"
}
```

Version 2.0 (Specification Proposal)

Procedure Call with positional parameters:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id":
1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id":
2}
<-- {"jsonrpc": "2.0", "result": -19, "id": 2}
```

Procedure Call with named parameters:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend":
23, "minuend": 42}, "id": 3}
<-- {"jsonrpc": "2.0", "result": 19, "id": 3}

--> {"jsonrpc": "2.0", "method": "subtract", "params": {"minuend": 42,
"subtrahend": 23}, "id": 4}
<-- {"jsonrpc": "2.0", "result": 19, "id": 4}
```

Notification:

```
--> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}

--> {"jsonrpc": "2.0", "method": "foobar"}
```

Procedure Call of non-existent procedure:

```
--> {"jsonrpc": "2.0", "method": "foobar", "id": 10}
<-- {"jsonrpc": "2.0", "error": {"code": -32601, "message": "Procedure
not found."}, "id": 10}
```

Procedure Call with invalid JSON:

```
--> {"jsonrpc": "2.0", "method": "foobar", "params": "bar", "baz"]
```

```
<-- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse
error"}, "id": null}
```

Procedure Call with invalid JSON-RPC:

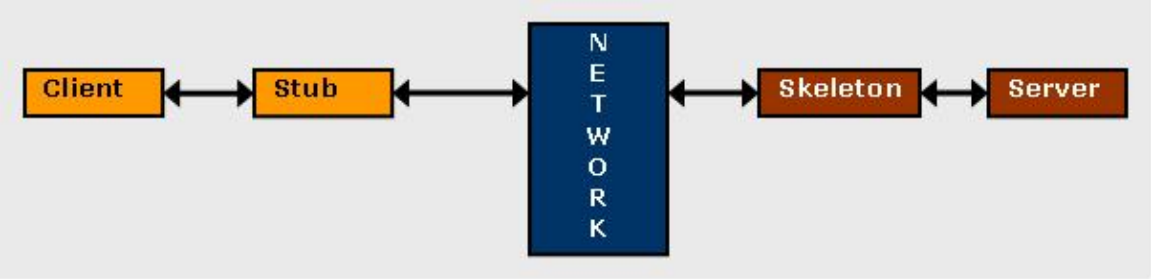
```
--> [1,2,3]
<-- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid
JSON-RPC."}, "id": null}

--> {"jsonrpc": "2.0", "method": 1, "params": "bar"}
<-- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid
JSON-RPC."}, "id": null}
```

Implementations

- jsonrpc4j is a java implementation JSON-RPC 2.0 supporting streaming as well as HTTP servers. It also has support for spring service exporter\consumer.
- json-rpc is a generic java/javascript implementation which integrates well on Android/Servlets/Standalone Java/Javascript/App-Engine applications.
- php-json-rpc is a simple PHP implementation of JSON-RPC 2.0.
- easyXDM is a library for cross-domain messaging with a built in RPC feature. The library supports all browsers by using a mix of postMessage, nix, frameElement, window.name, and FIM, and is very easy to use.
- Dojo Toolkit offers a broad support for JSON-RPC
- Pmrpc is an inter-window and Web Worker remote procedure call JavaScript library for use within HTML5 browsers. Pmrpc is an implementation of JSON-RPC using the HTML5 postMessage API for message transport.
- JSON Toolkit an implementation for Delphi
- Jayrock is a server implementation of JSON-RPC 1.0 for versions 1.1 and 2.0 of Microsoft's .NET Framework.
- As of version 2.0, XINS supports both JSON and JSON-RPC.
- qooxdoo includes a JSON-RPC implementation with optional backends in Java, PHP, Perl and Python.
- Barracuda Web Server's integrated JSON-RPC online documentation.
- JSON-RPC implementation in JavaScript includes JSON-RPC over HTTP and over TCP/IP Sockets
- JSON/XML-RPC Client and Server Implementations which abstract-away the differences between JSON-RPC and XML-RPC and permit cross-site requests.
- jabsorb - A lightweight Ajax/Web 2.0 JSON-RPC Java framework that extends the JSON-RPC protocol with additional ORB functionality such as circular references support.
- JSON-RPC implementation in Java A JavaScript to Java AJAX communications library (now merged with jabsorb.)
- Jettison - Java library
- LEWOStuff Includes Java support for JSON-RPC also with specific support for WebObjects.

- Perl implementations on the CPAN
- JsonRpc-Cpp OpenSource JSON-RPC implementation in C++
- TclSOAP Includes Tcl support for JSON-RPC in addition to SOAP and XML-RPC.
- Pyjamas contains a JSONRPC client implementation, as standard (Pyjamas is a framework where applications are written in Python but are compiled to Javascript).
- Zope 3 JSON-RPC Python based JSON RPC server and client implementation for Zope 3
- JQuery JSON-RPC Server This is a JSON-RPC server, specifically made to work with the Zend Framework JSON RPC Server. The Zend Framework JSON-RPC server is mildly off spec, and therefore this may not work with other JSON-RPC servers.
- jsonrpc2php is a PHP5 BSD'd JSON-RPC 2.0 Core class and example Server
- jsonrpclib is a JSON-RPC client module for Python.
- Tornado web server supports serving JSON-RPC using the tornadorpc plugin.
- JSON-RPC 2.0 Base is a minimal Java implementation of version 2.0 of the JSON-RPC protocol (open source).
- JSON-RPC 2.0 Shell for querying, testing and debugging remote JSON-RPC 2.0 web services (commercial).
- Synopse SQLite3 database Framework Open Source Delphi database access, User Interface generation, reporting, i18n in Client/Server AJAX/RESTful model.
- Objective-C DeferredKit includes a JSON-RPC 1.0 client.
- java-json-rpc JSON-RPC 2.0 implementation for J2EE servers.
- lib-json-rpc JSON-RPC 2.0 implementation on servlet, client, javascript
- simplejsonrpc Another simple JSON-RPC 2.0 Servlet, servicing the methods of a class.

Java remote method invocation



A typical implementation model of Java-RMI using stub and skeleton objects. Java 2 SDK, Standard Edition, v1.2 removed the need for a skeleton.

The **Java Remote Method Invocation** Application Programming Interface (API), or **Java RMI**, is a Java application programming interface that performs the object-oriented equivalent of remote procedure calls (RPC).

1. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP).
2. In order to support code running in a non-JVM context, a CORBA version was later developed.

Usage of the term **RMI** may denote solely the programming interface or may signify both the API and JRMP, whereas the term RMI-IIOP (read: RMI over IIOP) denotes the RMI interface delegating most of the functionality to the supporting CORBA implementation.

The programmers of the original RMI API generalized the code somewhat to support different implementations, such as a HTTP transport. Additionally, the ability to pass arguments "by value" was added to CORBA in order to support the RMI interface. Still, the RMI-IIOP and JRMP implementations do not have fully identical interfaces.

RMI functionality comes in the package `java.rmi`, while most of Sun's implementation is located in the `sun.rmi` package. Note that with Java versions before Java 5.0 developers had to compile RMI stubs in a separate compilation step using **`rmic`**. Version 5.0 of Java and beyond no longer require this step.

Jini offers a more advanced version of RMI in Java. It functions similarly but provides more advanced searching capabilities and mechanisms for distributed object applications.

Example

The following classes implement a simple client-server program using RMI that displays a message.

`RmiServer` **class**—Listens to RMI requests and implements the interface which is used by the client to invoke remote methods.

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.*;

public class RmiServer extends UnicastRemoteObject
    implements RmiServerIntf {
    public static final String MESSAGE = "Hello world";

    public RmiServer() throws RemoteException {
    }
```

```
    public String getMessage() {
        return MESSAGE;
    }


    public static void main(String args[]) {
        System.out.println("RMI server started");

        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Security manager installed.");
        } else {
            System.out.println("Security manager already exists.");
        }

        try { //special exception handler for registry creation
            LocateRegistry.createRegistry(1099);
            System.out.println("java RMI registry created.");
        } catch (RemoteException e) {
            //do nothing, error means registry already exists
            System.out.println("java RMI registry already exists.");
        }

        try {
            //Instantiate RmiServer
            RmiServer obj = new RmiServer();

            // Bind this object instance to the name "RmiServer"
            Naming.rebind("//localhost/RmiServer", obj);

            System.out.println("PeerServer bound in registry");
        } catch (Exception e) {
            System.err.println("RMI server exception:" + e);
            e.printStackTrace();
        }
    }
}
```

`RmiServerIntf` class—Defines the interface that is used by the client and implemented by the server.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RmiServerIntf extends Remote {
    public String getMessage() throws RemoteException;
}
```

`RmiClient` class—This is the client which gets the reference to the remote object and invokes its method to get a message.

```
import java.rmi.Naming;
import java.rmi.RemoteException;
```

```
import java.rmi.RMISecurityManager;

public class RmiClient {
    // "obj" is the reference of the remote object
    RmiServerIntf obj = null;

    public String getMessage() {
        try {
            obj =
(RmiServerIntf)Naming.lookup("//localhost/RmiServer");
            return obj.getMessage();
        } catch (Exception e) {
            System.err.println("RmiClient exception: " + e);
            e.printStackTrace();

            return e.getMessage();
        }
    }

    public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        RmiClient cli = new RmiClient();

        System.out.println(cli.getMessage());
    }
}
```

Before running this subj, we need to make 'Stub' file of interface we used. For this task we have RMI compiller - 'rmic'

- Note: we make stub file from *.class with implementation remote interface, not '*.java'*

```
rmic RmiServer
```

`server.policy`—This file is required on the server to allow TCP/IP communication for the remote registry and for the RMI server.

```
grant {
    permission java.net.SocketPermission "127.0.0.1:*",
"connect,resolve";
    permission java.net.SocketPermission "127.0.0.1:*", "accept";
};
```

The server.policy file should be used using the D switch of Java RTE, e.g.:

```
java.exe -Djava.security.policy=server.policy RmiServer
```

client.policy—This file is required on the client to connect to RMI Server using TCP/IP.

```
grant {
    permission java.net.SocketPermission "127.0.0.1:*",
"connect,resolve";
};
```

no.policy—Also if you have any troubles with connecting, try this file for server or client.

```
grant {
    permission java.security.AllPermission;
};
```

Chapter-13

XML-RPC and RPyC

XML-RPC

XML-RPC is a remote procedure call (RPC) protocol which uses XML to encode its calls and HTTP as a transport mechanism. "XML-RPC" also refers generically to the use of XML for remote procedure call, independently of the specific protocol.

History

XML-RPC, the protocol, was created in 1998 by Dave Winer of UserLand Software and Microsoft. As new functionality was introduced, the standard evolved into what is now SOAP.

The generic use of XML for remote procedure call (RPC) was patented by Phillip Merrick, Stewart Allen, and Joseph Lapp in April 2006, claiming benefit to a provisional application filed in March 1998. The patent is assigned to webMethods, located in Fairfax, VA.

Usage

XML-RPC works by sending a HTTP request to a server implementing the protocol. The client in that case is typically software wanting to call a single method of a remote system. Multiple input parameters can be passed to the remote method, one return value is returned. The parameter types allow nesting of parameters into maps and lists, thus larger structures can be transported. Therefore XML-RPC can be used to transport objects or structures both as input and as output parameters.

Identification of clients for authorization purposes can be achieved using popular HTTP security methods. Basic access authentication is used for identification, HTTPS is used when identification (via certificates) and encrypted messages are needed. Both methods can be combined.

In comparison to REST, where *resources* are transported, XML-RPC is designed to *call methods*.

XML-RPC is simpler to use and understand than SOAP because it

- allows only one method of method serialization, whereas SOAP defines multiple different encodings
- has a simpler security model
- does not require (nor support) the creation of WSDL service descriptions, although XRDL provides a simple subset of the functionality provided by WSDL

JSON-RPC is similar to XML-RPC.

Data types

Common datatypes are converted into their XML equivalents with example values shown below:

Name	Tag Example	Description
array	`<array><data><value><i4>1404</i4></value><value><string>Something here</string></value><value><i4>1</i4></value></data></array>`	Array of values, storing no keys
base64	`<base64>eW91IGNhbid0IHJlYWQgdGhpcyE=</base64>`	Base64-encoded binary data
boolean	`<boolean>1</boolean>`	Boolean logical value (0 or 1)
date/time	`<dateTime.iso8601>19980717T14:08:55</dateTime.iso8601>`	Date and time in ISO 8601 format
double	`<double>-12.53</double>`	Double precision floating point number
integer	`<i4>42</i4>`	Whole number,

		integer
	or <int>42</int>	
string	<string>Hello world!</string>	String of characters. Must follow XML encoding.
struct	<struct> <member> <name>foo</name> <value><i4>1</i4></value> </member> <member> <name>bar</name> <value><i4>2</i4></value> </member> </struct>	Associative array
nil	<nil/>	Discriminated null value; an XML-RPC extension

Examples

An example of a typical XML-RPC request would be:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
        <value><i4>40</i4></value>
    </param>
  </params>
</methodCall>
```

An example of a typical XML-RPC response would be:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

A typical XML-RPC fault would be:

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

Criticism

Critics of XML-RPC argue that RPC calls can be made with plain XML, (effectively this means REST) and that XML-RPC doesn't add any value over XML. Both XML-RPC and XML require an application level data model, such as which field names are defined in the XML schema or the parameter names in XML-RPC. Furthermore, XML-RPC uses about 4 times the number of bytes compared to plain XML to encode the same objects, which is itself bloated compared to JSON.

Implementations

Python

- xmlrpclib
  - Renamed xmlrpc.client in Python 3.
- Creating XML-RPC Servers and Clients with Twisted

C++

- Libiqxmlrpc
- Ultra lightweight XML-RPC library for C++
- XML-RPC for C and C++
- XmlRpc++
- XmlRpc C++ client for Windows
- gSOAP toolkit for C and C++ supporting XML-RPC and more
- libmaia: XML-RPC for Qt/C++

Objective-C / GNUstep / Cocoa

- XMLRPC Framework
- Cocoa XML-RPC Framework: Open Source XML-RPC framework written for use in Mac OS X Cocoa applications.

Erlang

- XML-RPC for Erlang: This is an HTTP 1.1 compliant XML-RPC library for Erlang. It is designed to make it easy to write XML-RPC Erlang clients and/or servers.

Java

- Apache XML-RPC: Open source library for Java
- XML-RPC Delight: Convenient serialisation/deserialisation for Apache XML-RPC using Java Annotations and Beans
- : Secure Apache XML-RPC
- Redstone XML-RPC Library: Redstone's Open Source Library - XML-RPC implementation in Java
- XML-RPC Library for Java ME: Open source client-side library for Java ME

XMPP

- pyJabberXMLRPC: Python classes for XMPP
- Jabber-RPC: Over the Extensible Messaging and Presence Protocol protocol

Perl

- RPC::XML: A set of Perl classes for core data, message and XML handling
- XML::RPC: Perl module implementation

Other

- JSON/XML-RPC Client and Server: Abstract away the differences between JSON-RPC and XML-RPC
- RemObjects SDK Delphi and .NET package for XML-RPC, in addition to SOAP and others
- RealThinClient SDK: For Delphi/C++
- XML-RPC for ActionScript: For Flash ActionScript 2.0
- as3-rpclib: For Flex/Actionscript 3
- XML-RPC.NET: Open source library for .NET clients and servers
- XmlRpc-Light: Client and server library for OCaml
- S-XML-RPC: Client and server library for Common Lisp
- clj-xmlrpc: XML-RPC client for Clojure
- necessary-evil: XML-RPC Client and Ring-based Server for Clojure

- PHP-XML-RPC: For PHP
- crVCL PHP Framework: Fast PHP Framework with native XML-RPC library
- HaXR: Client and server library for Haskell
- xi library with PHP and Javascript XML-RPC: For PHP and Javascript
- Ruby XML-RPC library: For Ruby
- XML-RPC interface to Lua: For Lua
- android-xmlrpc: A light XML-RPC client for Google Android
- XML-RPC for Tcl: A Tcl implementation of XML-RPC providing client and server support
- : RebXR, a full client/server XML-RPC implementation for REBOL.

RPyC

RPyC

Developer(s)	Tomer Filiba
Initial release	17 December 2005
Stable release	3.1.0 / March 21, 2011; 16 days ago
Development status	Active
Written in	Python
Operating system	Cross-platform
Type	Remote Procedure Call
License	MIT License

RPyC or **R**emote **Py**thon **C**all, is a python library for remote procedure calls (RPC), as well as distributed computing. Unlike regular RPC mechanisms, such as ONC RPC, CORBA or Java RMI, RPyC is transparent, symmetrical, and requires no special decoration or definition languages. Moreover, it provides programmatic access to any pythonic element, be it functions, classes, instances or modules.

Features

- Symmetrical—there is no difference between the client and the server—both can serve. The only different aspect is that the client is usually the side that initiates the action. Being symmetrical, for example, allows the client to pass callback functions to the server.
- Transparent—remote objects look and behave the same as local objects would.
- Exceptions propagate like local ones
- Allows for synchronous and asynchronous operation:
    - Synchronous operations return a *NetProxy*
    - Asynchronous operations return an AsyncResult, which is like promise objects
    - AsyncResults can be used as events
- Threads are supported (though not recommended)
- UNIX specific: server integration with inetd

Architecture

RPyC gives the programmer a slave python interpreter at his or her control. In this essence, RPyC is different than other RPCs, that require registration of resources prior to accessing them. As a result, using RPyC is much more straight-forward, but this comes at the expense of security (you cannot limit access). RPyC is intended to be used within a trusted network, there are various schemes including VPN for achieving this.

Once a client is connected to the server, it has one of two ways to perform remote operations:

- The *modules* property, that exposes the server's modules namespace: `doc = conn.modules.sys.path` or `conn.modules["xml.dom.minidom"].parseString("<some>xml</some>").`
- The *execute* function, that executes the given code on the server: `conn.execute("print 'hello world'")`

Remote operations return something called a *NetProxy*, which is an intermediate object that reflects any operation performed locally on it to the remote object. For example, conn.modules.sys.path is a NetProxy for the sys.path object of the server. Any local changes done to conn.modules.sys.path are reflected immediately on the remote object. Note: NetProxies are not used for *simple objects*, such as numbers and strings, which are immutable.

Async is a proxy wrapper, meaning, it takes a NetProxy and returns another that wraps it with asynchronous functionallity. Operations done to an AsyncNetProxy return something called AsyncResult. These objects have a '.is_ready' predicate, '.result' property that holds the result (or blocks until it arrives), and '.on_ready' callback, which will be called when the result arrives.

Usage

Originally, RPyC was developed for managing distributed testing of products over a range of different platforms (all capable of running python). However, RPyC has evolved since then, and now its use cases include:

- Distributed computing (splitting workload between machines)
- Distributed testing (running tests that connect multiple platforms and abstracting hardware resources)
- Remote administration (tweaking config files from one central place, etc.)
- Tunneling or chaining (crossing over routable network boundaries)

Demo

```
import rpyc
conn = rpyc.classic.connect("hostname")  # assuming a classic server is
running on 'hostname'

print conn.modules.sys.path
conn.modules.sys.path.append("lucy")
print conn.modules.sys.path[-1]

# a version of 'ls' that runs remotely
def remote_ls(path):
    ros = conn.modules.os
    for filename in ros.listdir(path):
        stats = ros.stat(ros.path.join(path, filename))
        print "%d\t%d\t%s" % (stats.st_size, stats.st_uid, filename)

remote_ls("/usr/bin")

# and exceptions...
try:
     f = conn.builtin.open("/non/existent/file/name")
except IOError:
     pass
```

History

RPyC is based on the work of Eyal Lotem (aka Lotex) on PyInvoke, which is no longer maintained. The first public release was 1.20, which allowed for symmetrical and transparent RPC, but not for asynchronous operation. Version 1.6, while never publicly released, added the concept of 'events', as a means for the server to inform the client. Version 2.X, the first release of which was 2.2, added thread synchronization and the *Async* concept, which can be used as a superset of events. Version 2.40 adds the *execute* method, that can be used to execute code on the other side of the connection directly. RPyC 3 is a complete rewrite of the library, adding a capability-based security model, explicit services, and various other improvements.