ANOTHER CLIENT/SERVER-GlUE

CARSTEN PITZ
Carsten Pitz, Deutsche Post IT-AEZ, Moltkestr. 21-23, 54292 Trier, Germany
C.Pitz@DeutschePost.de

Abstract
Building, as well as running, large high availability client server systems is a complex task. Middleware is praised to be the solution for all these problems. But todays middleware, no matter wether message queue systems, message brokers, transaction monitors, object request brokers or object transaction monitors, lack some essential concepts to master this complexity.

Another weak point is data security: cryptographic methods to establish a connection, as well as to encrypt a data stream, are often insufficient or even missing.

In this paper a highly modular approach of a client/server-glue is introduced, which aims to target these problems.

Keywords:
distributed system, middleware, maintenance

1. MOTIVATION

Middleware is the glue to stick together distributed computer systems into a single system. So, at the first glance, it is a means of communication. This is – obviously – the main task, but just the tip of the iceberg. A middleware can – as described later on – offer you more.

There are two models for communication, referred to as synchronous and asynchronous. The synchronous model is implicitly transactionally secure. After sending, the client (caller) waits for a reply from the server (callee). If the reply is an error condition, the client can retry. The synchronous communication model maps onto remote procedure call (RPC) or remote message invocation (RMI). It is realised by transaction monitors, object request brokers and object transaction monitors.

The main disadvantage of the synchronous approach is, the client is blocked until the reply has been received. A common solution – beside multithreaded clients – is asynchronous communication. In this model the client writes a message (request) into a queue of the server. A scheduler within the server decides – e.g. due to priorities – when this request will be served. So, the asynchronous communication is not implicitly transactionally secure. The common way here is, to persistently store requests within the client, until a positive acknowledgement has been received. A watchdog mechanism ensures periodical retries. The message queue approach, in its transactionally secure form, is – obviously – the native model for message queue systems as well as for message brokers. But it is also supported by most transaction monitors.
So, the communication in its purest form is well covered. But what about data security? This topic is a – more or less – dark side of today’s middleware. Some products (e.g. BEA Tuxedo [2], a transaction monitor) today offer buildin 40bit stream encipher, rather insufficient regarding today’s compute power, but no buildin functionality to establish a connection in a secure manner. Other products (e.g. IBM MQSeries [3], a message queue system) offer exits (plugs) for 3rd-party security products. An elegant solution, but one which boosts the TCO (total cost of ownership). A little more in-depth discussion of this aspect is done in section 3.

A real dark side is multiple version support. A service (e.g. a business object) can evolve [1]. So the interface of a service can change. This topic is covered in section 4.

Shadows also lay the last topic to discuss: the maintenance of a distributed system and what’s the middleware’s role in this topic. This is discussed in section 5.

But in front of all these aspects, to explain, why we want to use a middleware, a small sketch of our needs is drawn in the following section.

2. MIDDLEWARE, WHY?

The parcel service of the Deutsche Post has a hierarchical 3 level topology. As shown in figure 1, a single central service center, which provides central computer based services (e.g. central databases), is the top node. The middle layer consists of 33 parcel centers. Here a pre-sort of the parcels is done. The bottom layer consists of 480 service bases. Here the final sort of the parcels is done and from here the mailmen get their parcels to be delivered. Parcels are transported between parcel centers as well as between a parcel center and its service bases.

![Figure 1: Topology of the parcel service](image)

The work presented in this paper has been done at the system design group of the AEZ Trier. The AEZ Trier focuses on marketing and sales systems for the parcels division of the Deutsche Post. An analysis of the communication topologies of several applications resulted in the presented design.

---

1Dotted arrows refer to control flow, solid to parcel flow.
On a normal day some 2 million parcels are to be serviced. In burst times (e.g. christmas) some 2.5 million per day. Each parcel has an ident code and a transport code. These codes are realized by two bar code badges, which are scanned by the sorters as well as by the mailmen, using hand scanners. Beside this, the signatures of the addressees are scanned to document the deliveries.

The information collected by the scanners is the base for several services ranging from coordinating parcel flow to generating reports for the management.

A middleware
– in the context of this paper –
is a tool to harmonize information flow.

But,
how does a middleware look like,
which fits our needs?

An object request broker (namely CORBA) would nicely fit the OO approach used by Deutsche Post software developers, but has too much overhead, when used to shift large amounts of data. Remote method invocation (RMI) on the other hand is too basic. But our information flow can be perfectly mapped onto a message queue system and the message queue approach can be elegantly embedded in an OO design (serialization). So, the message queue approach has been choosen.

Figure 2: The design: In/Out as separate operations

Beside the barcode and signature scanners, all imaginable information sources should be adaptable. Also, several different sources should be able to send simultaneously. The same should be true for data sinks as well as for application logics.
This implies, that the core of the middleware presented here must not have any knowledge about the semantics of the information transported. In the suggested design data sources, data sinks and application logics are plugins. The interplugin communication within a single instance of the middleware is – like the interinstance communication – itself based on messages.

In the present concept messages can be orders. These orders are atomic, so the task to be performed to fulfill an order itself is atomic. The sources are active, they send an order to an application logic, as soon as they have one, and the sinks are passive, they wait for an order to fulfill.

So, the middleware (figure 2²) can be characterized as an event driven, concurrent system. Consequently an input scheduler is needed to either statically or dynamically limit the input stream, to protect the system against temporal overload.

An application logic (Application Logic) or a passive sink (Out) is considered to be a serving task, a server. In an event driven, concurrent system servers elegantly can be instantiated and destroyed on the fly. And that is the way this concept handles it.

On the other hand, an active source (In) is a listening task, a listener or – using the client/server terminology – a client. The clients are the actors. So, they must neither be instanciated nor be destroyed by a middleware instance.

3. **DATA SECURITY**

Up to now, just the way how to get information in/out an instance and how application logics are integrated has been covered. The next point to discuss obviously is, how to transport information between instances.

Information transport between instances – generally – is done over networks. These networks can be public. Since the information to be transported (e.g. signatures) can be confidential, data security concerns.

Like the data sources/sinks and application logics, all imaginable data security mechanisms should be adaptable. So, also here a plugin-mechanism is used.

In this concept, establishing a connection (connect/accept) and transporting the information (data in/data out) are distinct, separated operations, performed by distinct plugins. The connect-plugin is, like an In-plugin, a listener. The other plugins are servers.

4. **MULTIPLE VERSION SUPPORT**

A service can – or even sometimes must – evolve. Additionally a service sometimes is parameterized. The middleware has to provide concepts to handle these two aspects.

On the last year’s DCW workshop, John Plaice presented a set of concepts [1] to master these aspects. The middleware presented here, is based on some of these concepts.

A service is specified by

²Dotted arrows refer to control flow, solid to data flow. Components beginning with a capital must “know” about semantics, those beginning with a small letter must not.
Figure 3: The design: inter-instance communication

- name,
- major release,
- minor release,
- patch level,
- variant (optional).

To order a service, a client has to specify

- name,
- major release,
- minor release,
- variant (optional).

This implies, that a patch must not change the protocol (interface). Changing the interface must be documented at least by a change in the minor release.

The present approach allows a server to concurrently provide several versions of several variants of a service. Simultaneously it allows to update a specific version of a variant of a service without modifying its clients.
Variants of the service "data transport" for example might be a data transport with or without data encryption. Other services might be country dependent, so one might have to implement an English, a French and a German variant.

The middleware itself is based on the client/server principle. The data out plugin for example provides a service for the data out demultiplexer. So, the versioning is also to be used inside the middleware itself.

5. MAINTENANCE

Versioning combined with self-contained components is one mayor key to ease maintenance. But two further keys are still missing.

![Diagram](image)

**Figure 4: The design: logging & remote control**

The first is, the updated components must be transported to and installed on the machines, which run these components. The old, no longer used components must be deinstalled.

The transport of information – e.g. software components – is a core functionality of this middleware, but installing and deinstalling software components is not, or not so far. One possibility is to provide an Application Logic plugin for this task. But, it is a generic task and not a business task. So, a more clean solution is, to provide this functionality as a core
functionality of the middleware. Additionally, to completely separate specific business tasks from generic middleware tasks, a separate channel must exist. As seen in figure 4 this results in additional control in and control out plugins as well as a kernel.

The second key missing is, there must be some information, what has to be maintained, where the weak parts are. So, some mechanism to collect and gather these information has to be implemented. In this design each instance has its own private logbook, realized by a logging plugin. All plugins and components (multiplexer, demultiplexer and scheduler) – via the kernel – as well as the kernel itself write their logs into this logbook. To be able to combine logs of several distinct, distributed instances the timestamps must match. This results in the need for synchronous clocks or a time service. This time service, like the install/deinstall service, is generic and consequently also a kernel component.

But one mechanism to spot the weak parts is still missing. What, if an instance crashes so hard, that it cannot even log it? A watchdog mechanism is needed. A watchdog on a specific instance periodically orders dummy services on all or some directly reachable instances and verifies the results. If it detects an error, it logs the error. This watchdog service (watchdog) is the third kernel service beside the install/deinstall service (loader) and the time service (time service).

![Diagram of the kernel design](image)

Figure 5: The design: the kernel

6. CONCLUSIONS

The design presented so far, is it a middleware? This question is a rather complex one. Yes, you can build a middleware based on this design. But it is more or less just a bucket of LEGO™ building bricks, the plugins, and a “base plate”, which is divided into kernel, multiplexers, demultiplexers and schedulers. You can cover (almost?) every imaginable topology using this design. You also can just use a subset of the components described in the concept. E.g. in a top node of a hierarchical topology you might leave away the whole accept, data in and control in part including the associated multiplexers. So, even the “base plate” is not fixed.

All information is transformed into a byte stream by the In and Application Logic plugins. It is transformed back to its original form by the Out and Application Logic plugins. This scheme – familiar from Java – is commonly called serialization/deserialization and is a simple, but powerful way to decouple the semantics from the design. At the first glance, a major drawback
of this scheme – when used on top, not within a programming language – seems to be, that it leaves the responsibility of handling machine dependent formats (e.g. big/little endian) to the application developers, for the associated In and Out plugins are part of an application. But a generic representation layer covering the In-, Out- and Application Logic-plugins can eliminate this problem.

REFERENCES

