Architectural Styles for Distribution

Using macro-patterns for system design

Charles Weir

Abstract

This paper highlights the problem of describing the software architecture of a distributed system, and introduces the Architectural Styles proposed by Shaw&Garlan as a possible solution. Using a pattern template, it explores four major styles for distribution architecture: Host-Terminal, Client-Server, Broadcast Data and Batch Communication.

Introduction

One major problem we find in building large software systems is the problem of talking about their structure. How do we discuss different ways of putting together system components? Currently we lack a sufficiently rich vocabulary.

There is a very effective method to build up a vocabulary. We use narrative, examples, similes and references to build up new concepts in the mind of the reader, and then give this concept a name. From then on we can use the name in our discussions and documentation. I am aware of two leading researchers using this kind of approach: [Jackson] describes 'Problem Frames' (structures for a system), and uses a narrative approach to describe them and the architectural issues surrounding them. [Shaw&Garlan] describe 'Architectural Styles', using approach similar to that of the pattern movement.

This paper extends the work of [Shaw&Garlan] to explore styles for the architecture of distributed systems.

Shaw&Garlan's Architectural Styles

[Shaw&Garlan] distils the essence of large system design using an approach very similar to the software patterns first promoted by Richard Gabriel and derived from the work of Christopher Alexander. As with a pattern, each *architectural style* has a simple short descriptive name. However the authors do not use a template for their styles. Instead a narrative description attempts to cover the following items:

A short name to use in conversation.
What is the design vocabulary (the components and connectors)?
What are the allowable structural patterns?
What is the underlying computational model?
What are the essential invariants of the style?
What are common examples of its use?
What are the advantages and disadvantages of using that style?
What are some of the common specialisations?

Table 1: Elements of a Shaw&Garlan Architectural Style

Using Styles

The book [Shaw&Garlan] goes on to stress that in practice any given system will use a combination of styles, rather than exclusively one. In this respect, architectural styles appear very similar to software patterns ([Gamma+]).

I myself have used styles to discuss the overall software architecture for two new software projects. I find, as with patterns, the names and the rules of behaviour implied by each style helps considerably when sketching out options for the high-level design of a system.

However the descriptions in [Shaw&Garlan] are limited. Their narrative approach leads to some being explained well; others sketched only in the vaguest detail. My experiments with describing styles in this informal way (see [Weir]) have led me to conclude that a more structured approach, based on the pattern format, is more profitable. This paper uses such a pattern structure to describe several styles commonly used in distribution architectures.

Architectural styles differ significantly in content from many of the other types of software pattern, so this paper uses a rather different pattern template. The template preserves the elements of the Shaw&Garlan styles (see Table 1) as follows:

Item	Purpose
Name	A short name to use in conversation.
The Problem	Where is the style useful?
The Solution	What is the design vocabulary (the components and connectors)? What are the allowable structural patterns and the underlying computational model?
Invariants	What are the rules for any system conforming to this style?
Applicability	What are the advantages and disadvantages of using that style in specific situations?
Examples	What are common examples of its use?
Variations	How is it possible to vary the style for specific situations?

Table 2: The pattern format used in the descriptions.

I've found in practice that perhaps the most important of these is the discussion of the *invariants* of each style. These provide rules for developers implementing the architectures; we know that we are keeping to a specific architectural style if we haven't broken its invariants.

Styles for Distribution Architecture

This paper introduces four commonly used styles,

1. The Host-Terminal Style
2. The Client-Server Style
3. The Broadcast Data Style
4. The Batch Communication Style

The following sections describe these in detail.

The Host-Terminal Style

The Problem

In very many systems we have a number of users or data entry points which are geographically distributed, but which share common resources such as a database. In many systems there has been much less computing power needed or available in the distributed systems than in the central system.

The Solution

In the mainframe-terminal style we leverage the limited processing available in the distributed components by making them run a single, fixed, program: the *terminal*. This terminal can then be optimised to make the best possible use of the limited resources in each component. The terminal supports a single *terminal protocol*, and has a single logical connection directly to the *host*, which is shared between all of the terminals.

Thus the components are: the terminals and the host. The connectors are: the terminal protocols.

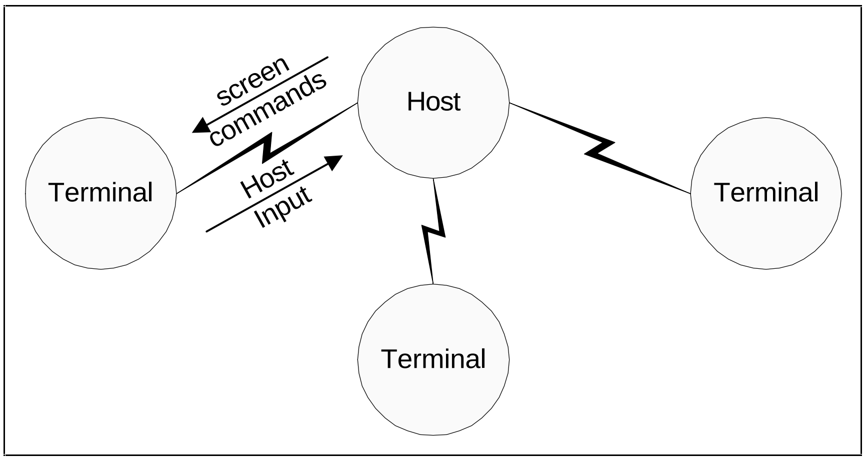

Figure 1: The Host-Terminal Style

Typically a terminal will not have facilities dedicated to any specific application. Instead typical examples of facilities supported by terminals are:

- Rendering text at an arbitrary point on the screen
- Rendering graphics on the screen
- Rendering and supporting GUI components, such as buttons or menus, on the screen.
- Sending user keystrokes, menu selections, and pointer events to the host
- Creating 'forms' according to a template. These allow the terminal to 'batch' up user input locally for more efficient processing by the host.
- Support printer output sent from the host to a printer connected to the terminal.

Invariants

The terminal communicates only with the host.

The terminal runs a single software program, which may interface to one or many different programs on the host.

The terminal software is controlled completely by the host program.

Applicability

This style is suitable for applications supporting multiple users, which can run on a single host.

All the significant application processing takes place in the host. This means that we need not take communication delays and problems into consideration when designing the application software.

However the host-terminal style is effective only when:

1. There is a continuous real-time link between terminal and host.
2. Either the user interface is relatively simple, or the bandwidth of the link is large and the host has plenty of processing power to devote to each user interface.

The cost of providing both of the above can be considerable, particular given the needs of modern users for easy-to-learn, and therefore intelligent and graphical, user interfaces.

Examples

The Host-terminal style has its roots in the earliest forms of shared computer system. Even today this style is probably the most common form of distributed processing. Often the terminals involved nowadays themselves will be PCs, capable of much more powerful processing in their own right. However the terminal protocols (and terminal emulator software) remain in many cases the only practical way to communicate with mainframe systems.

The DEC VT100 Terminal. This character-based terminal displays characters on the screen and passes user keystrokes back to the host. The simplicity of the protocol model has made it a popular approach for mini-computers, and there are many variants of this protocol.

The IBM 370 terminal. These character-based terminals use forms to reduce the load on the host and the communication bandwidth required.   This makes them more suitable than the VT100 terminals for mainframes supporting large numbers of terminals.

Variations

The X Terminal extends the terminal concept to support a graphical user interface via a local area network (LAN) or Wide Area Network (WAN). It supports communication with multiple simultaneous hosts. In fact, an X Terminal uses the 'Client-Server Architectural Style' (see below) to share its resources between these multiple hosts. It acts as a server supplying screen and keyboard interface resources that are shared between many client applications, who use RPC protocols to interface to the server.

For form-based data entry, it is frequently not necessary for the user to receive acknowledgement that the form has been processed before continuing. So some form-based terminals support intermediate 'batching systems' between the terminal and the host. These take the data entry from many terminals and batch it into single large messages for efficient processing by the mainframes, while still ensuring data integrity. These batching systems appear as hosts to the terminals.

The Client-Server Style

The Problem

Often we have situations where much of the processing can be local to specific machines, and yet we need some co-ordination between them. For example, modern PCs can provide significant local graphical user interface support, and some simple data validation. Yet if the users are to interact with any kind of enterprise-wide or global data set, they will need access to shared resources. And these shared resources will need to be managed carefully to avoid conflicts as more than one system attempts to access them.

The Solution

The client-server style provides a solution. In a client-server style the components involved are *Clients* and *Servers*.   They communicate via a logical connection that lasts the duration of the interaction. Each client makes *requests* to a server, which generates *responses* back.

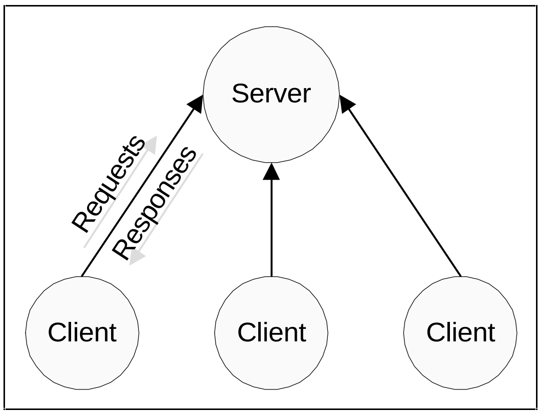

Figure 2: The Client-Server Model

The clients and servers are roles; a given component may take the role both of client and server. These components may be variously:

- Processes

- Objects, or
- Hardware devices

Invariants

A component has no accessible state other than that revealed by its *client interface.*

The server has no a-priori knowledge of any given client. It is the responsibility of each client to locate and ‘register with’ the server.

Clients have no communication between themselves, other than via the server. Although a client of one server may itself be a server to other clients.

Applicability

The client-server model is suitable for architectures where:

1. Both clients and server command significant processing power.
2. There is a clear separation between the roles of the client and the server. And each interaction is driven by a requester, the client, and a supplier, the server.
3. There is not too much shared data between the client and the server. To much data or too complex an interface will make the communication between them unwieldy.
4. The server has no need for a-priori knowledge of its clients.

A widely-used mechanism for client-server interactions is *Remote Procedure Calls (RPC).*

The chief benefit of the style is that the underlying model is easy to understand. An RPC implementation maps intuitively onto the call-and-return semantics of most common programming languages. Similarly the object broker standards extend these semantics to match object-oriented languages.

However there can be a significant penalty in client complexity and performance. The complexity occurs because both client and server must be designed with distribution in mind; the interface between them is subject to significant restrictions, particularly that each call takes a significant time.

The performance penalty occurs because of the time for the network round-trip and the task switch times involved. In particular, processing at the client must wait for the return of the procedure. To get around this, some client-server interfaces allow functions that return no data to return immediately, without waiting for the server message round-trip. Where the server reply is required, the client must implement all the complexities of multi-threading in order to provide a response to other stimuli while waiting for a server reply.

Also the thread aspects can become complicated: typically a client thread will need to block until it receives a response. In other words, the thread effectively spans two components, the client and the server. If the server is itself a client of further servers, this thread may continue in further components, perhaps even back to the original client. This sometimes requires a multi-threaded environment in the client, the server, or frequently both.

Some client interfaces (such as Sun’s NFS protocols) are state-less; the server behaves as though it has no retained information about any client between each RPC. More commonly, client interfaces are session-based; the server retains information of the client’s *session.*

A stateless model is more difficult to design and implement. However it can make it much easier to optimise the performance of the server, since there is no requirement to preserve data locks between client calls. Since data locks can become the main bottleneck in large-scale systems, this improvement is significant. [Waldo+] provides a further discussion of the problems of state-based distribution.

Examples

The CORBA standard and the Microsoft COM system (see [Orfali+]) are both based on a client-server model. Here the components are objects that may be distributed over a network.

The DCE standard uses a client-server model to provide interprocess communication across a network.

Forte and Dynasty are development environments using the client-server model. They both allow developers to produce code on a system with no distribution, then add the distribution as part of the deployment of the finished system.

Many common distribution mechanisms are based on the client-server model. For example:

Databases	Distributed databases such as Sybase and Oracle use client-server processes. The server processes manage the database; client processes contain the user code. A typical RPC call contains a series of SQL instructions.
File Server	Distributed file server protocols, such as Sun's NFS or Novel's NetWare, use the client-server model.
GUI	The X Window system uses a client-server approach. Here the server renders the images on the screen; the clients are the X applications.

Many of the Internet client protocols (Telnet, FTP and HTTP, for example) use a client-server model. HTTP, in particular, uses a stateless model, which partially explains the success of the World Wide Web. The phenomenal performance of such servers as the Web Search engines can occur only because they have no need to preserve client state.

Variations

In some environments the client-server model is extended to support more than one response to a single request. For example, the client may register with the server, and subsequently receive data broadcast from the server. Or the client may make calls that require no return value, and may therefore be sent and processed asynchronously. Or else the client may provide its own identity and an interface to the server as part of the registration, allowing the server to make call-backs to the client when necessary.

The client-server style is also common in situations where there is no physical distribution at all, but where client and server are merely in separate processes. In this case, the style provides a 'firewall' between the client code and the server code. Microsoft's OCX components use a client-server model, but have only recently started to support physical distribution. Similarly a vast majority of the systems using IBM/Apple OpenDoc environment have client and server physically co-located, even though the CORBA protocol allows them not to be.

The Broadcast Data Style

The Problem

Sometimes we need to distribute information to many points, where only the receivers, not the senders, need to ensure the integrity of the data. This situation is common in news and real-time financial trading systems.

The Solution

In the Broadcast Data Style, a *server* broadcasts messages that may be received by multiple *receivers*. The messages conform to a protocol that allows the receivers to detect any missing messages.

So the components are:	Servers and receivers.
The connectors are:	A broadcast stream of messages.

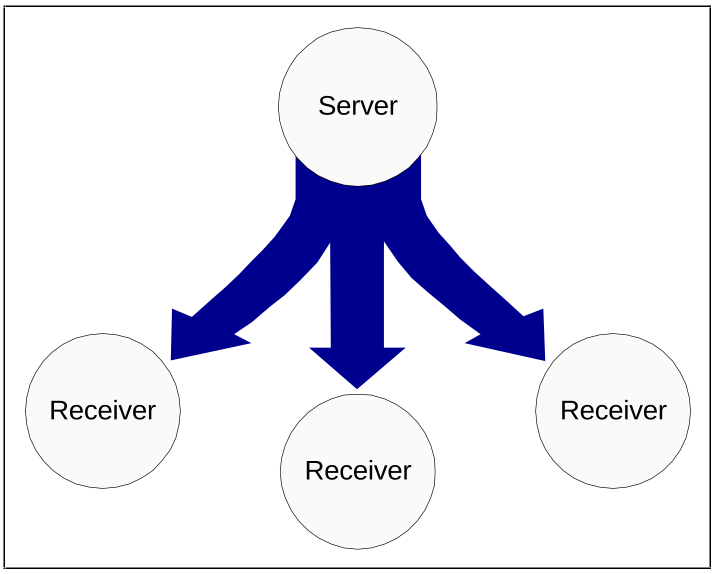

Figure 3: The Broadcast Data Style

In most LANs and WANs, the receivers must first set up the broadcast channel, by notifying the server which data streams they wish to receive. A common approach is to use the Client-Server style (see page 4).

The Broadcast data style is the distributed analogue of 'Event System' style in [Shaw&Garlan]. The broadcast messages correspond to events in the Event System style.

Invariants

The broadcast stream is one-way, from the server to any number of receivers.

The server has no interest in whether any specific receiver has received a particular message. It is the sole responsibility of the receivers to ensure their data integrity. The protocol must include mechanisms to allow this.

Applicability

The broadcast data style is suitable for systems where one process is aware of events that must be communicated to a number of others.

Usually the broadcast protocol will include a mechanism for the receivers to re-retrieve lost data. This may be by a direct interaction with the server (usually using the Client-Server mechanism), or with a periodic re-broadcast of the data by the server.

The style is obviously well suited to exploiting hardware broadcast mechanisms such as satellite broadcast, LAN broadcast, and LAN multicast protocols. However it may also be appropriate in situations where there is a two-way link, such as a network connection or a serial line, between the server and every receiver. The benefit of the style in this case is that the protocol makes few demands on the server; in most cases the server need not manage separate connections to each receiver but can treat its output mechanism as a simple pipe.

Examples

Reuters International Data Network (IDN) provides the world-wide dissemination of real-time financial data. Many of the protocols used conform to the Broadcast Data style. In particular, the satellite links, both between major financial centres and from the data centres to client terminals, use broadcast mechanisms with no communication at all from the receiver to the server. IDN also includes LAN-based protocols such as the Triarch's RRCP, where the receiver does request specific broadcast streams and can re-request missed messages.

Another financial distribution mechanism, the Teknekron TIB bus, also uses a LAN-based protocol where the receiver makes requests and can re-request lost data.

The Internet Usenet protocols use broadcast mechanisms, where the receiver can request specific broadcast streams.

The Prestel system uses television signals to broadcast text data to special receivers associated with the television.

There are several extensions to the Internet HTTP protocol and the CORBA ORB protocols to allow broadcast distribution of information over the Internet. An example is the proprietary 'pointcast' protocol for distributing web pages.

The Batch Communication Style

The Problem

The previous three styles all rely on a continuous electronic link between all of the distributed systems. However in many cases such a link may not exist, or may not provide sufficient bandwidth to allow anything approach a real-time connection for all of the applications that need to communicate.

The Solution

In the Batch communication style one component creates messages that are processed another at a later time. We call the first component the *sender* and the second the *receiver.* Between the two, there is a *store and forward* protocol, which ensures the eventual delivery of data between the two.

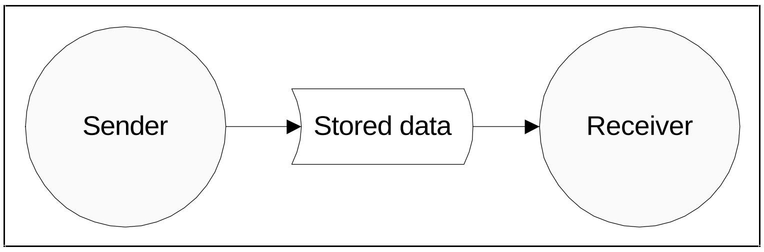

Figure 4: The Batch Communication Style

Invariants

The sender system assumes that the data will be delivered. The infrastructure takes responsibility of handling delivery problems, time-outs and lost messages; these are not the responsibility of the sender.

The system preserves some kind of ordering of messages between a specific sender and receiver - although this need not necessarily be a strict sequence. If the sequencing is not strict, it is up to the sender to indicate which data must be sequenced.

Applicability

This style is appropriate for any distribution architecture where the sender process does not need direct confirmation of any actions taken by the receiver process.

It is commonly used where real-time communication is expensive or impossible.

It is also usually much simpler to implement than a real-time connection between two systems and therefore has smaller development costs.

There are many different possible delivery mechanisms underlying the protocol, for example:

- Transaction-based store and forward mechanisms such as IBM's MQ system. These guarantee delivery or notify a human operator of the problem.

- Lightweight store and forward mechanisms, such as the Email protocols or the Usenet NNTP protocol.
- Writing to file store and transferring the resulting file, either electronically via protocols such as FTP, or physically as tape, disk, or CD.

The different mechanisms have different performance, cost and speed attributes.

Examples

The 'batch processing' idiom common to mainframe systems uses this style. Typically in batch processing, of course, the sender and receiver are separate in time, rather than geographically distributed; however geographic distribution (often implemented by carrying tapes around) is also common.

A more recent invention, 'Message-Oriented-Middleware', implements a similar scheme for mainframe-style systems allowing systems to communicate over a network using asynchronous messages whose delivery is guaranteed. Examples include IBM's MQ system.

Both Lotus Notes and the Usenet Network News Transfer Protocol (NNTP) use batch communication to implement a batch update from one server to another (known as 'replication'). Each server acts as both sender and receiver (although not, usually, at the same time); they send new database records they have received from their users and other servers.

Most Email systems use the batch communication style. Here the senders and receivers are the creators and readers of the mails; the intermediate protocols are message systems such as the Simple Mail Transfer Protocol (SMTP).

The terminal 'batching systems' discussed in 'The Client-Server Style' use a batch communication style. In this case the senders are the terminals; the receivers are the mainframe's form processing systems.

Other Related Styles

The above are only a few of many possible styles for distribution architecture. Some alternatives are:

- Peer-to-peer communication. Many host systems send one another messages as peers.

  This style is suitable where the system must continue functioning even when a single component is missing or broken. For example the TCP Internet protocol (and other modern network bridge systems) is based on peer-to-peer communication between network routers who are all equal.

- Inter-process communication. Within a single system, applications may perform inter-process communication using shared memory and semaphores.

  This style is suitable where very fast communication is required between the processes on a single host.

- Shared database systems. The distributed systems use client-server facilities provided by their database vendor. From the point of view of the application designer, the approach is the Database style of [Shaw&Garlan].

  This style is suitable for applications that need data consistency between the different components. Examples include airline reservation or sales entry systems.

References

[Coplien&Schmidt] *Pattern Languages of Programming*, Coplien and Schmidt, Addison-Wesley, ISBN 0-201-60734-4

[Gamma+]	*Design Patterns*, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, ISBN 0-201-63361-2
[Jackson]	*Software Requirements and Specifications*, Michael Jackson, Addison-Wesley, ISBN 0-201-87712-0
[Orfali+]	*Essential Distributed Objects Survival Guide*, Orfali, Harkey and Edwards, John Wiley and Sons Ltd., ISBN 0-471-12993-3
[Shaw&Garlan]	*Software Architecture - Perspectives on a Emerging Discipline,* Shaw and Garlan, Prentice Hall ISBN 0-13-182957-2
[Waldo+]	*A Note on Distributed Computing*, Waldo, Wyant, Wollrath and Kendall, Sun Microsystems Laboratories, http://www.sunlabs.com/smli/technical-reports/1994/smli_tr-94-29.ps
[Weir]	*A Client-Server Architectural Style*, Charles Weir, Submission to Working Group at Object Technology 97. http://www.cix.co.uk/~cweir/ot97wg/papers/cli-ser.ps