DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN
By: Dr. Faramarz Safi
Islamic Azad University
Najafabad Branch

Chapter 2
ARCHITECTURES

Architectural Styles (1)

Distributed systems are often complex pieces of software of which the components are by definition dispersed across multiple machines. To master their complexity, it is crucial that these systems are properly organized. There are different ways on how to view the organization of a distributed system, but an obvious one is to make a distinction between the logical organization of the collection of software components and on the other hand the actual physical realization.

Important styles of architecture for distributed systems

- Layered architectures
- Object-based architectures
- Data-centered architectures
- Event-based architectures

Architectural Styles (2)

The basic idea for the layered style is simple: components are organized in a layered fashion where a component at layer *L; is allowed to call components at* the underlying layer *Li, but not the other way around, as shown in Fig. 2-1(a).* This model has been widely adopted by the networking community; we briefly review it in Chap.4. An key observation is that control generally flows from layer to layer; requests go down the hierarchy whereas the results flow upward.

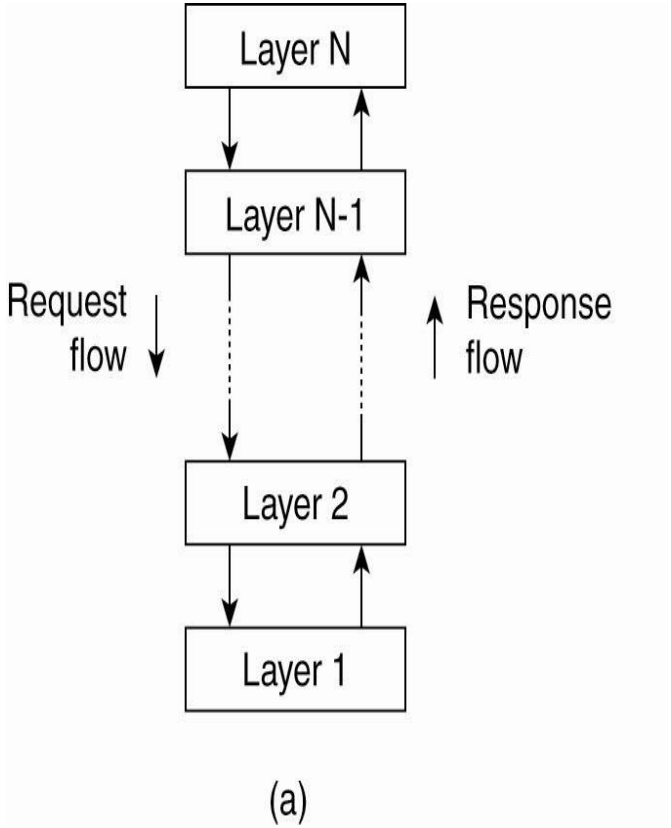


Figure 2-1. The (a) layered architectural style and …

Architectural Styles (3)

A far looser organization is followed in object-based architectures, which are illustrated in Fig. 2-1(b). In essence, each object corresponds to what we have defined as a component, and these components are connected through a (remote) procedure call mechanism. Not surprisingly, this software architecture matches the client-server system architecture. The layered and object-based architectures still form the most important styles for large software systems (Bass et aI., 2003).

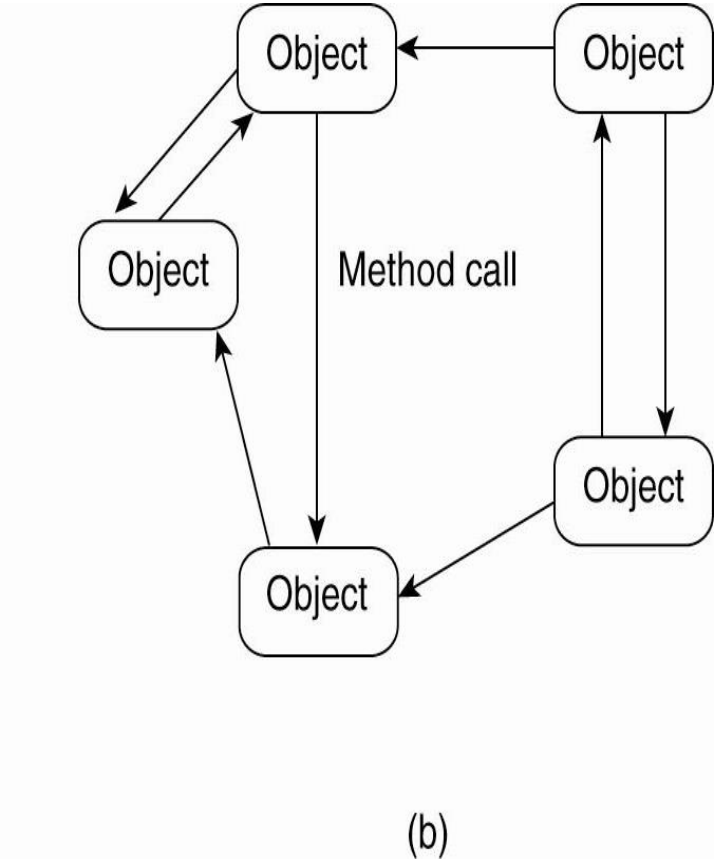


Figure 2-1. (b) The object-based architectural style.

Architectural Styles (4)

In event-based architectures, processes essentially communicate through the propagation of events, which optionally also carry data, as shown in Fig. 2-2(a).

For distributed systems, event propagation has generally been associated with what are known as publish/subscribe systems (Eugster et aI., 2003). The basic idea is that processes publish events after which the middleware ensures that only those processes that subscribed to those events will receive them. The main advantage of event-based systems is that processes are loosely coupled. In principle, they need not explicitly refer to each other. This is also referred to as being decoupled in space, or referentially decoupled.

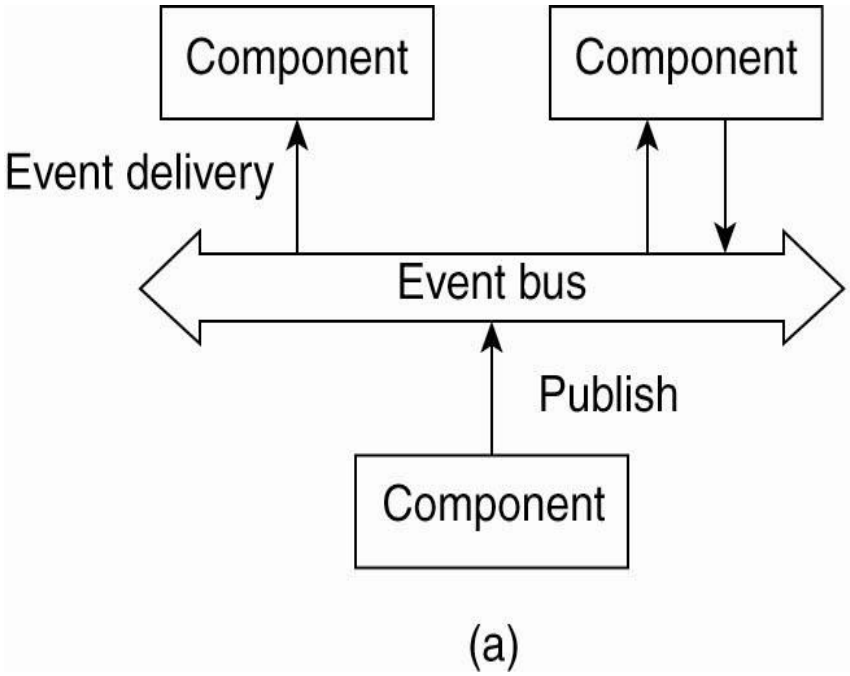


Figure 2-2. (a) The event-based architectural style and …

Architectural Styles (5)

Data-centered architectures evolve around the idea that processes communicate through a common (passive or active) repository. It can be argued that for distributed systems these architectures are as important as the layered and object-based architectures. For example, a wealth of networked applications have been developed that rely on a shared distributed file system in which virtually all communication takes place through files. Likewise, Web-based distributed systems are largely data-centric: processes communicate through the use of shared Web-based data services.

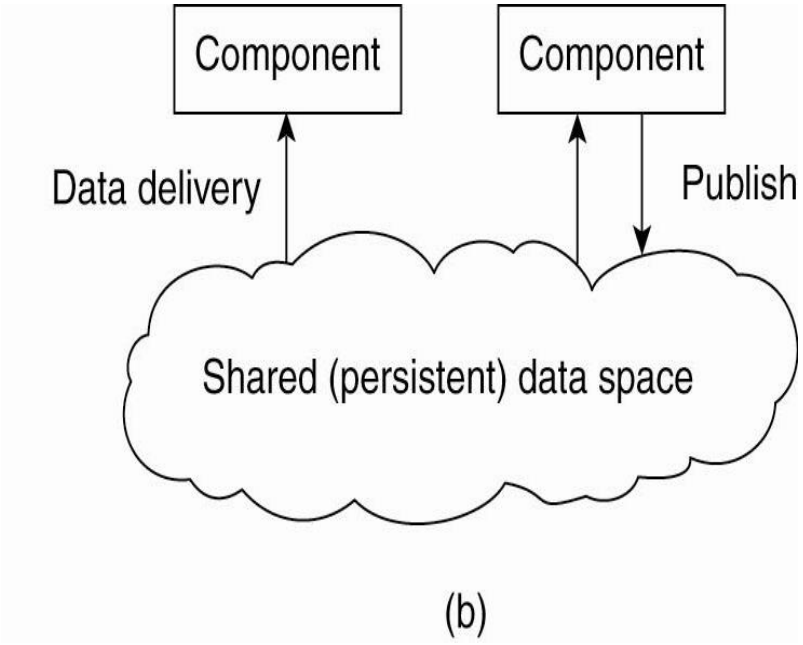


Figure 2-2. (b) The shared data-space architectural style.

System Architecture

There are three views toward system architectures:

- Centralized Architectures
- Multi-tierd Architectures
- Distributed Architectures

Multi-tiered Architectures
Two tier

The simplest organization is to have only two types of machines:

- A client machine containing only the programs implementing (part of) the user-interface level
- A server machine containing the rest,
  - the programs implementing the processing and data level

Multi-tiered Architectures

Two tier

In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service or a database service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as **request-reply behavior** is shown in Fig. 2-3. Connection less (UDP) and Connection oriented methods (TCP) are used to implement this communication. The trouble is that setting up and tearing down a connection is relatively costly, especially when the request and reply messages are small. The main issues is how to draw a clear distinction between a client and a server.

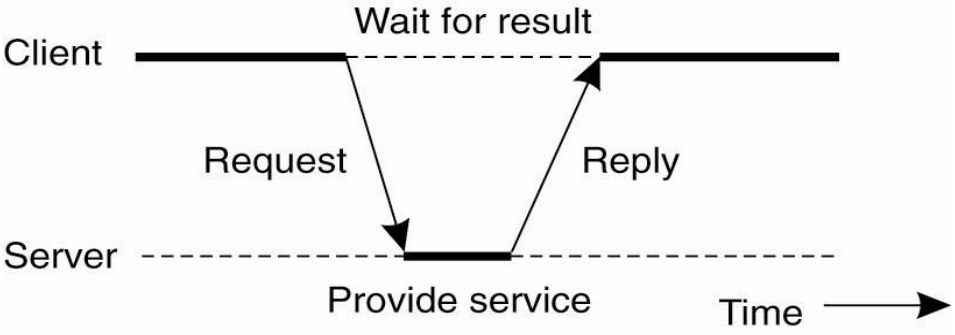


Figure 2-3. General interaction between a client and a server.

Multi-tiered Architectures
Three tier

Three Layers of architectural style

- The user-interface level
- The processing level
- The data level

Decision Support Systems, and Desktop packages are some more examples

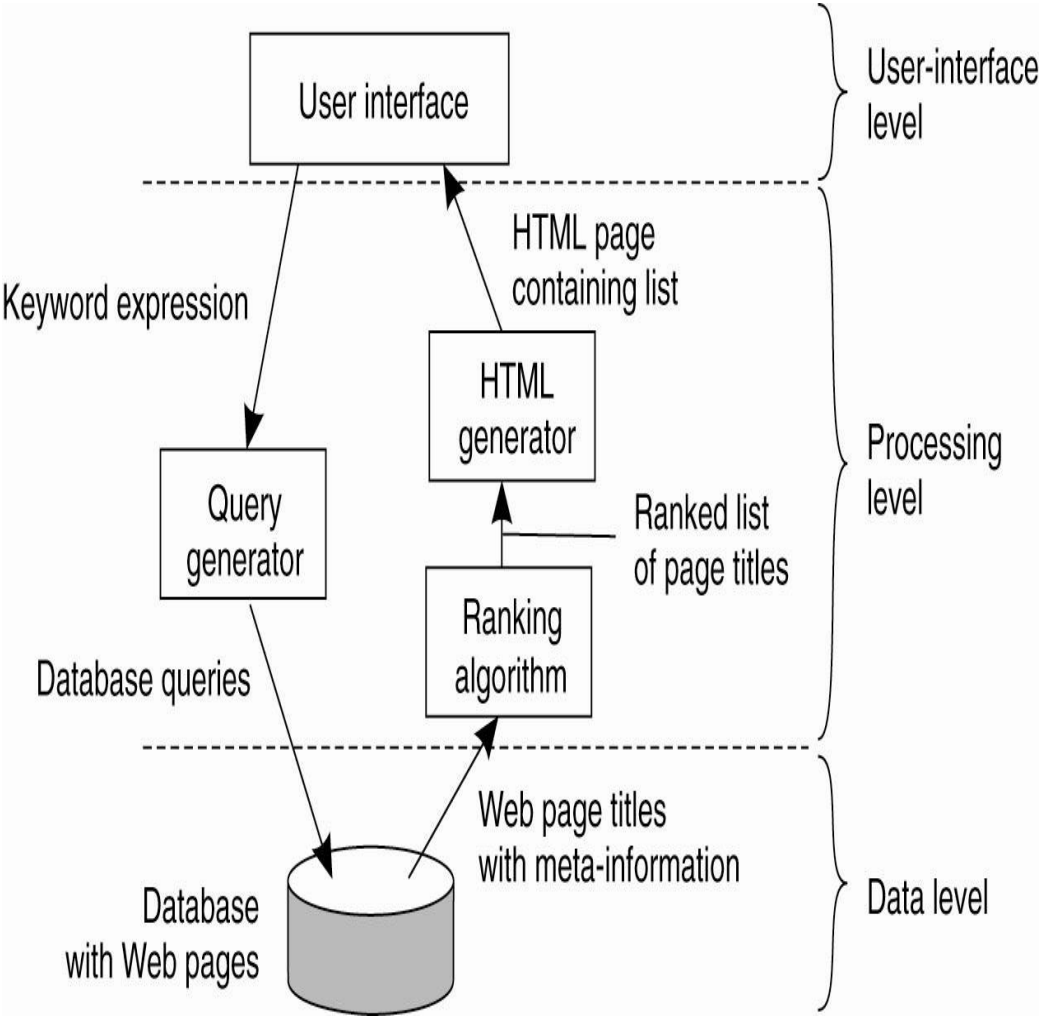


Figure 2-4. The simplified organization of an Internet search engine into three different layers.

Multi-tiered Architectures

چه قسمتی از Application را به Server بدهیم. اگر کل داده ها روی Client باشد خود تبدیل به سرور می شود.

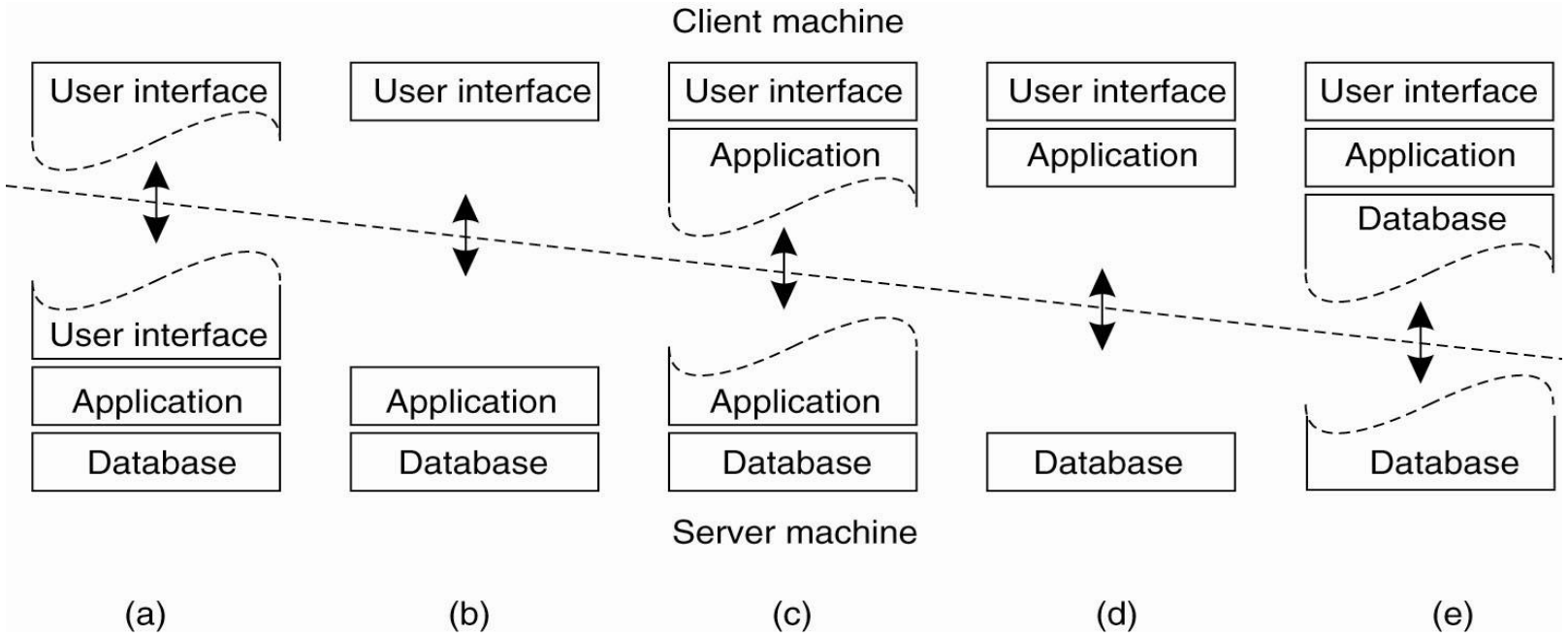


Figure 2-5. Alternative client-server organizations (a)–(e).

Multi-tiered Architectures
three-tier to n-tier

در معماری های سه تکه ای تا n تکه ای، server ها خود Client لایه های دیگر می باشند. بعنوان نمونه ای از سیستم های n تکه ای می توان از .Net، J2EE و CORBA نام برد. سیستم های پردازش تراکنش و سازماندهی وب سایت ها از دیگر مثال های این معماری سه لایه می باشند.

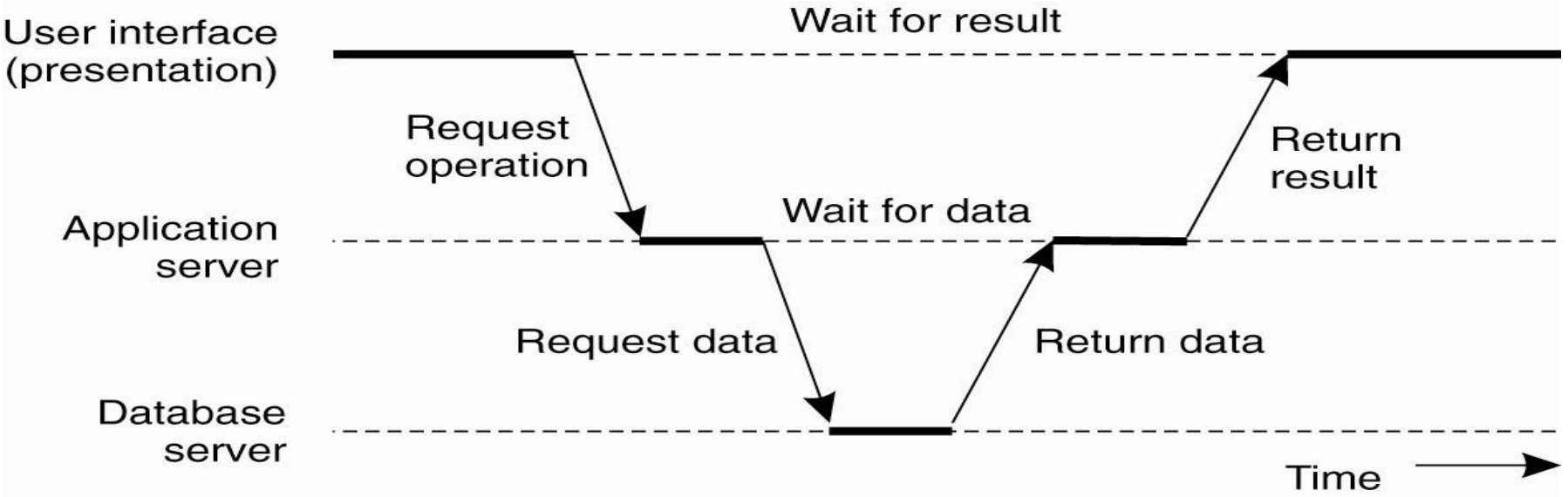


Figure 2-6. An example of a server acting as client.

Multi-tiered Architectures
Vertical Distribution (n-tier)

Multi-tiered client-server architectures are a direct consequence of dividing applications into a user-interface, processing components, and a data level.

The different tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multi-tiered architecture. We refer to this type of distribution as vertical distribution which are n-tier applications.

The characteristic feature of vertical distribution is that it is achieved by placing *logically different components on different* machines.

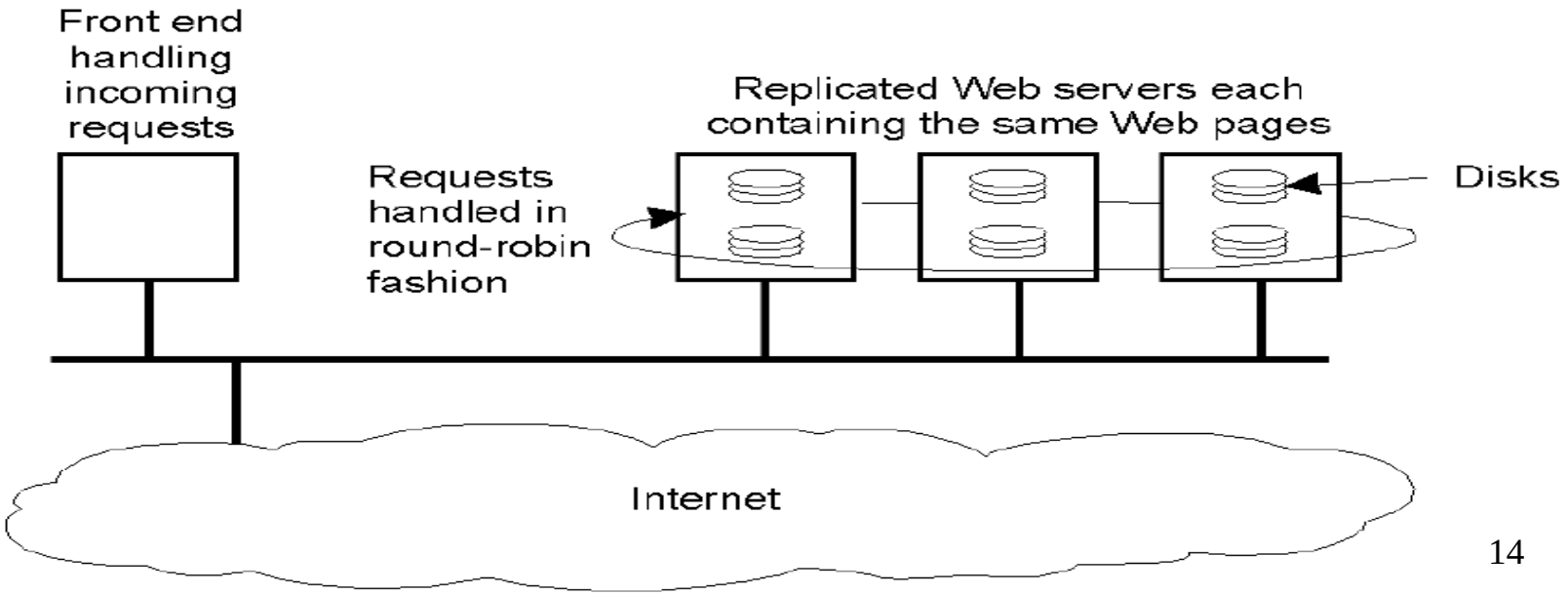
Again, from a system management perspective, having a vertical distribution can help:

Functions are logically and physically split across multiple machines, where each machine is tailored to a specific group of functions.

Decentralized Arcitectures
Horizontal Distribution (Modern Architectures)

In modern architectures (Horizontal Distribution) a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. A class of modern system architectures that support horizontal distribution, known as peer-to-peer systems.

از مزیت چنین سیستم هایی اشتراک بار (توازن بار)، قابلیت اطمینان و تحمل بالای خطا را می توان نامبرد. مثالی دیگر از این سیستم یک وب سرور می باشد که در شکل زیر نشان داده شده است.



Modern Architectures

Peer-to-Peer Architecture

From a high-level perspective, the processes that constitute a peer-to-peer system are all equal. This means that the functions that need to be carried out are represented by every process that constitutes the distributed system. As a consequence, much of the interaction between processes is symmetric: each process will act as a client and a server at the same time (which is also referred to as acting as a servant).

Given this symmetric behavior, peer-to-peer architectures evolve around the question how to organize the processes in an **overlay** network, that is, a network in which the nodes are formed by the processes and the links represent the possible communication channels (which are usually realized as TCP connections). In general, a process cannot communicate directly with an arbitrary other process, but is required to send messages through the available communication channels. Two types of overlay networks exist: those that are **structured** and those that are not (**unstructured**).

Hybrid Architectures

we take a look at some specific classes of distributed systems in which client-server solutions are combined with decentralized architectures. Some of these architectures are as follows:

- Edge-Server Systems
- Collaborative Distributed Systems

Hybrid Architectures
Edge-Server Systems

An important class of distributed systems that is organized according to a hybrid architecture is formed by edge-server systems. These systems are deployed on the Internet where servers are placed "at the edge" of the network. This edge is formed by the boundary between enterprise networks and the actual Internet, for example, as provided by an Internet Service Provider (ISP). Likewise, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of the Internet.

End users, or clients in general, connect to the Internet by means of an edge server. The edge server's main purpose is to serve content, possibly after applying filtering and transcoding functions. More interesting is the fact that a collection of edge servers can be used to optimize content and application distribution. The basic model is that for a specific organization, one edge server acts as an origin server from which all content originates. That server can use other edge servers for replicating Web pages

Hybrid Architectures
Edge-Server Systems

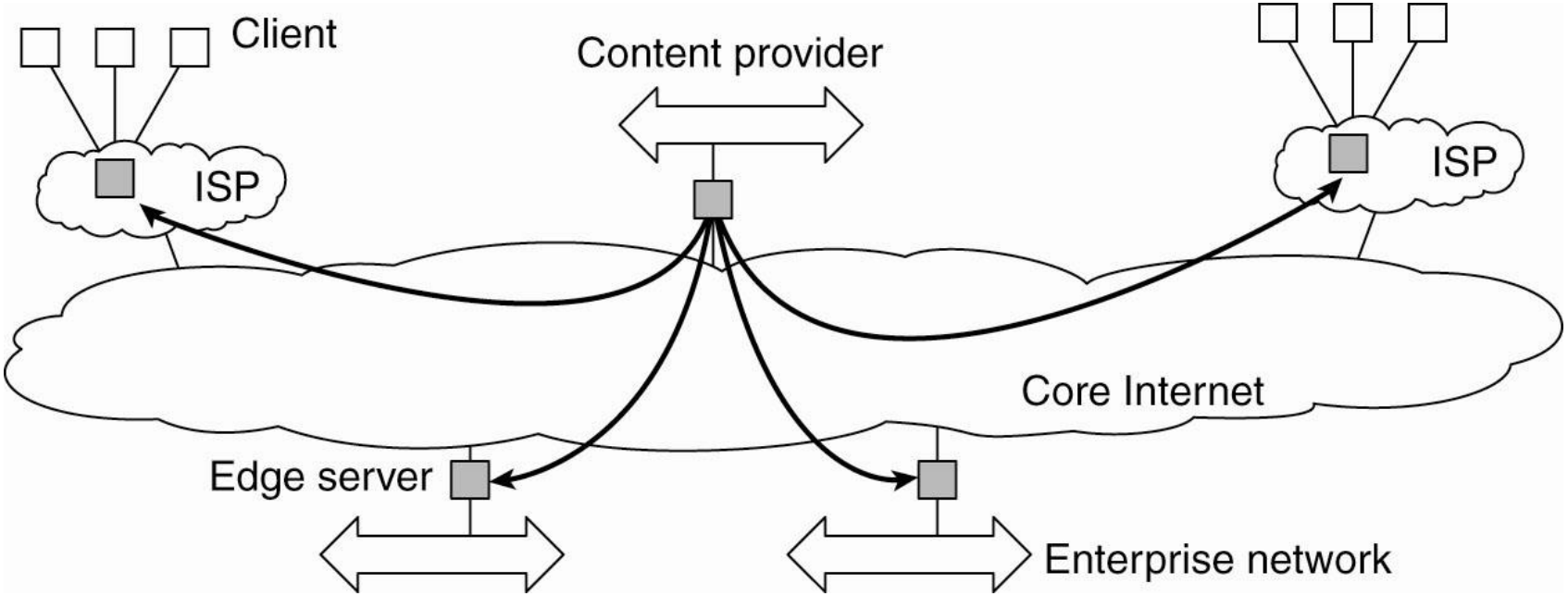


Figure 2-13. Viewing the Internet as consisting of a collection of edge servers.

Hybrid Architectures
Collaborative Distributed Systems (1)

Hybrid structures are notably deployed in collaborative distributed systems. The main issue in many of these systems is to first get started, for which often a traditional client-server scheme is deployed. Once a node has joined the system, it can use a fully decentralized scheme for collaboration.

BitTorrent is a **peer-to-peer** file downloading system. Its principal working is shown in Fig. 2-14. The basic idea is that when an end user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file. An important design goal was to ensure collaboration. In most file-sharing systems, a significant fraction of participants is merely download files but otherwise contribute close to nothing. To this end, a file can be downloaded only when the downloading client is providing content to someone else.

Hybrid Architectures
Collaborative Distributed Systems (2)

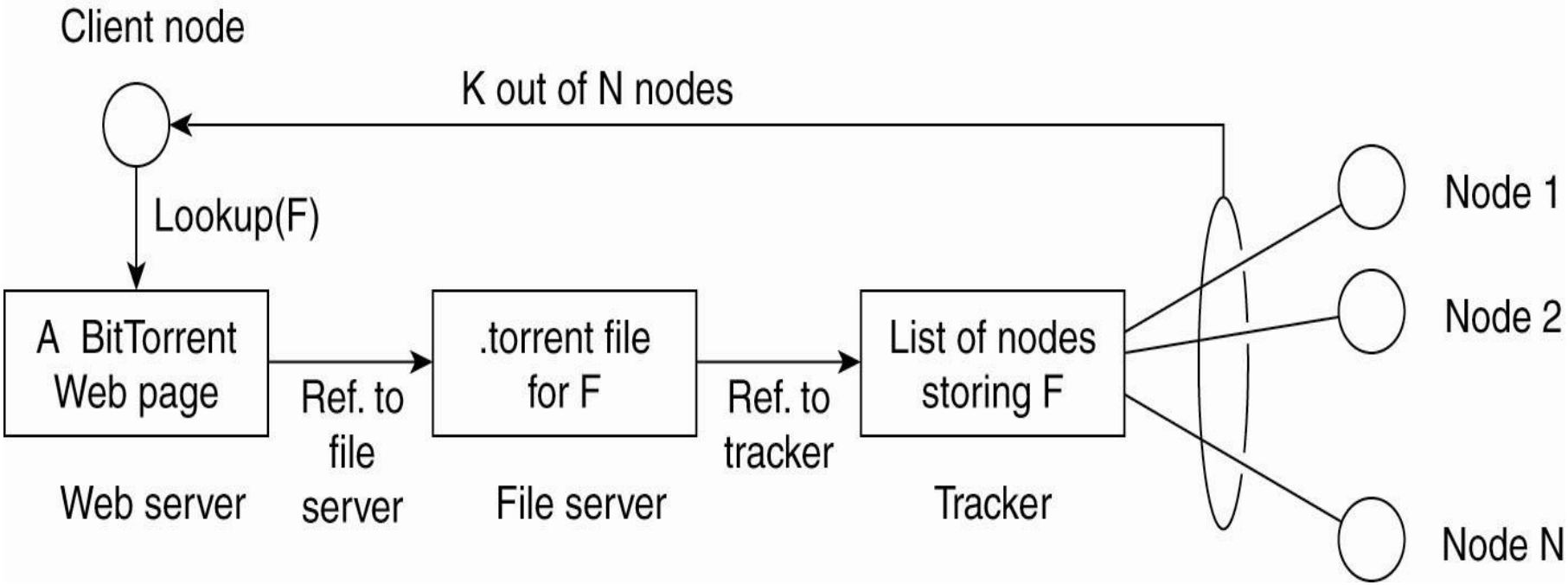


Figure 2-14. The principal working of BitTorrent [adapted with permission from Pouwelse et al. (2004)].

Hybrid Architectures
Collaborative Distributed Systems (3)

As another example, consider the Globule collaborative Content Distribution Network (CDN). Globule strongly resembles the edge-server architecture mentioned above. In this case, instead of edge servers, end users (but also organizations) voluntarily provide enhanced Web servers that are capable of collaborating in the replication of Web pages.

In its simplest form, each such server has the following components:

- A component that can redirect client requests to other servers.
- A component for analyzing access patterns.
- A component for managing the replication of Web pages.

Architectures Vs. Middleware

When considering the architectural issues we have discussed so far, a question that comes to mind is where middleware fits in. middleware forms a layer between applications and distributed platforms. As shown in Fig. 1-1. An important purpose is to provide a degree of distribution transparency, that is, to a certain extent hiding the distribution of data, processing, and control from applications.

What is commonly seen in practice is that middleware systems actually follow a specific architectural style. For example, many middleware solutions have adopted an object-based architectural style, such as CORBA. Others, like TIB/Rendezvous provide middleware that follows the event-based architectural style.

Advantage:

Having middleware molded according to a specific architectural style has the benefit that designing applications may become simpler.

Disadvantage:

The middleware may no longer be optimal for what an application developer had in mind.

Architecture Vs. Middleware Two Solutions

Although middleware is meant to provide distribution transparency, it is generally felt that specific solutions should be adaptable to application requirements.

One solution to this problem is to make several versions of a middleware system, where each version is tailored to a specific class of applications.

Another approach that is generally considered better is to make middleware systems such that they are easy to configure, adapt, and customize as needed by an application.

As a result, systems are now being developed in which a stricter separation between policies and mechanisms is being made. This has led to several mechanisms by which the behavior of middleware can be modified (Sadjadi and McKinley, 2003). **Using Interceptors and Adaptive software are two policies in the second solution.**

Architectures Vs. Middleware Interceptors

Conceptually, an **interceptor is nothing but a software construct that will break the usual flow of control and allow other (application specific) code to be executed.**

To make matters concrete, consider interception as supported in many object-based distributed systems. The basic idea is simple: an object *A can call a method* that belongs to an object *B, while the latter resides on a different machine than A.* As we explain in detail later in the book, such a remote-object invocation is carried as a three-step approach:

1. Object *A is offered a local interface that is exactly the same as the interface* offered by object *B. A simply calls the method available in* that interface. 2. The call by *A is transformed into a generic object invocation, made* possible through a general object-invocation interface offered by the middleware at the machine where *A resides.* 3. Finally, the generic object invocation is transformed into a message that is sent through the transport-level network interface as offered by *A's local operating system.*

Architectures Vs. Middleware Interceptors

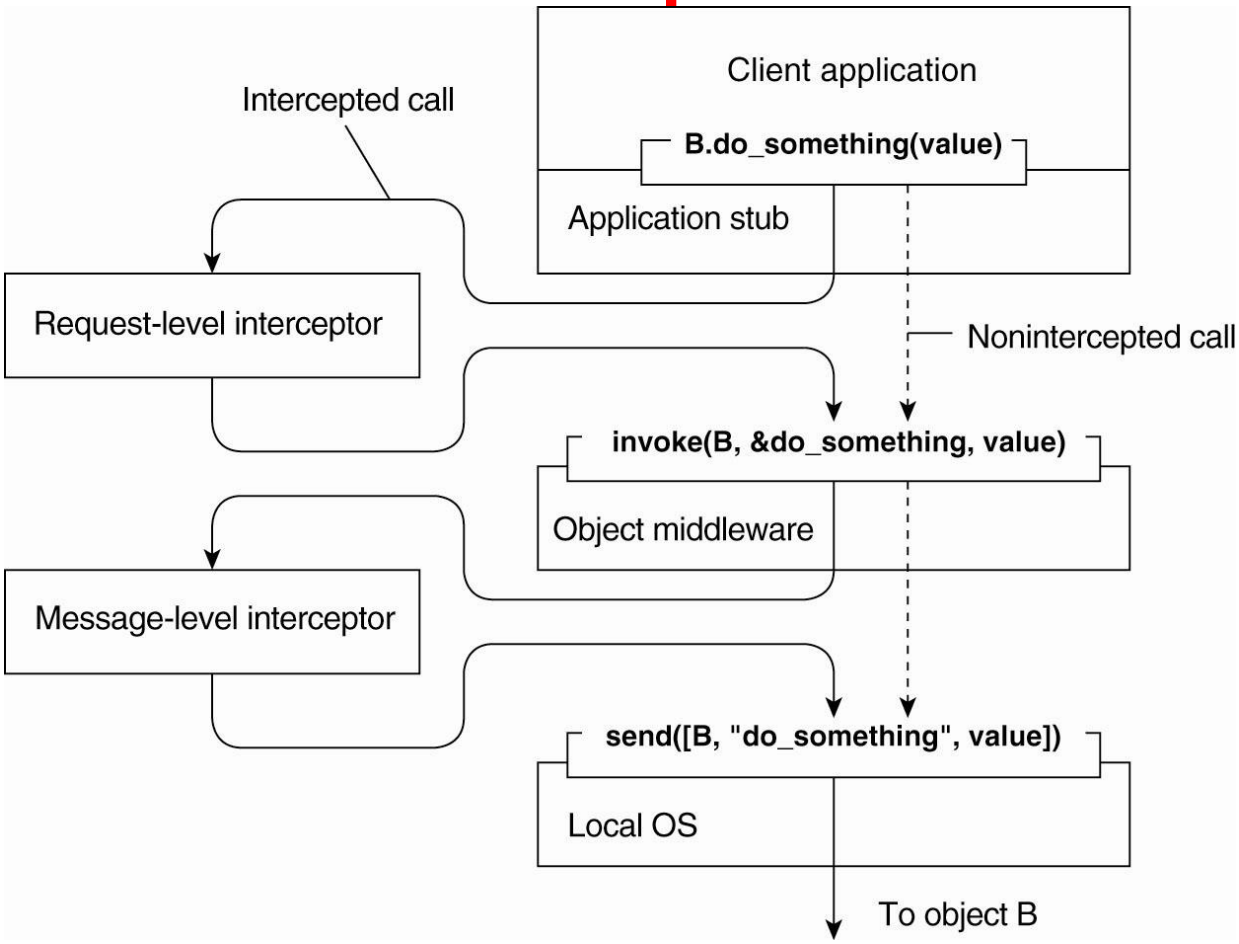


Figure 2-15. Using interceptors to handle remote-object invocations.

General Approaches to Adaptive Software

What interceptors actually offer is a means to adapt the middleware. The need for adaptation comes from the fact that the environment in which distributed applications are executed changes continuously. Changes include those resulting from mobility, a strong variance in the quality-of-service of networks, failing hardware, and battery drainage, amongst others. Rather than making applications responsible for reacting to changes, this task is placed in the middleware.

These strong influences from the environment have brought many designers of middleware to consider the construction of *adaptive software. However, adaptive* software has not been as successful as anticipated. As many researchers and developers consider it to be an important aspect of modern distributed systems, let us briefly pay some attention to it. There are three basic techniques to come to software adaptation:

- Separation of concerns
- Computational reflection
- Component-based design

Adaptive Software
Separation of Concerns

Separating concerns relates to the traditional way of modularizing systems:

separate the parts that implement functionality from those that take care of other things (known as *extra functionalities) such as reliability*, *performance*, *security*, etc. One can argue that developing middleware for distributed applications is largely about handling **extra functionalities** independent from applications. The main problem is that we cannot easily separate these extra functionalities by means of modularization.

For example, simply putting security into a separate module is not going to work. Likewise, it is hard to imagine how fault tolerance can be isolated into a separate box and sold as an independent service. Separating and subsequently weaving these *cross-cutting concerns into a (distributed) system* is the major theme addressed by **aspect-oriented software development**

Adaptive Software
Computational Reflection

Computational reflection refers to the ability of a program to inspect itself and, if necessary, adapt its behavior (Kon et al., 2002). Reflection has been built into programming languages, including Java, and offers a powerful facility for runtime modifications. In addition, some middleware systems provide the means to apply reflective techniques. However, just as in the case of aspect orientation, reflective middleware has yet to prove itself as a powerful tool to manage the complexity of large-scale distributed systems. Applying reflection to a broad domain of applications is yet to be done.

Adaptive Software
Component-based Design

Finally, component-based design supports adaptation through **composition**. A system may either be configured statically at design time, or dynamically at runtime. The latter requires support for late binding, a technique that has been successfully applied in programming language environments, but also for operating systems where modules can be loaded and unloaded at will. Research is now well underway to allow automatically selection of the best implementation of a component during runtime, but again, the process remains complex for distributed systems, especially when considering that replacement of one component requires knowing what the effect of that replacement on other components will be.

Self-* in Distributed Systems

In this section we pay explicit attention to organizing distributed systems as high-level feedback-control systems allowing automatic adaptations to changes. This phenomenon is also known as **autonomic computing** or Self-* (self.star) systems. The latter name indicates the variety by which automatic adaptations are being captured: self-managing, self-healing, self-configuring, self-optimizing, and so on. We resort simply to using the name self-managing systems as coverage of its many variants.

Autonomic Computing
The Feedback Control Model

There are many different views on self-managing systems, but what most have in common (either explicitly or implicitly) is the assumption that adaptations take place by means of one or more **feedback control loops**. Accordingly, systems that are organized by means of such loops are referred to as feedback Control systems.

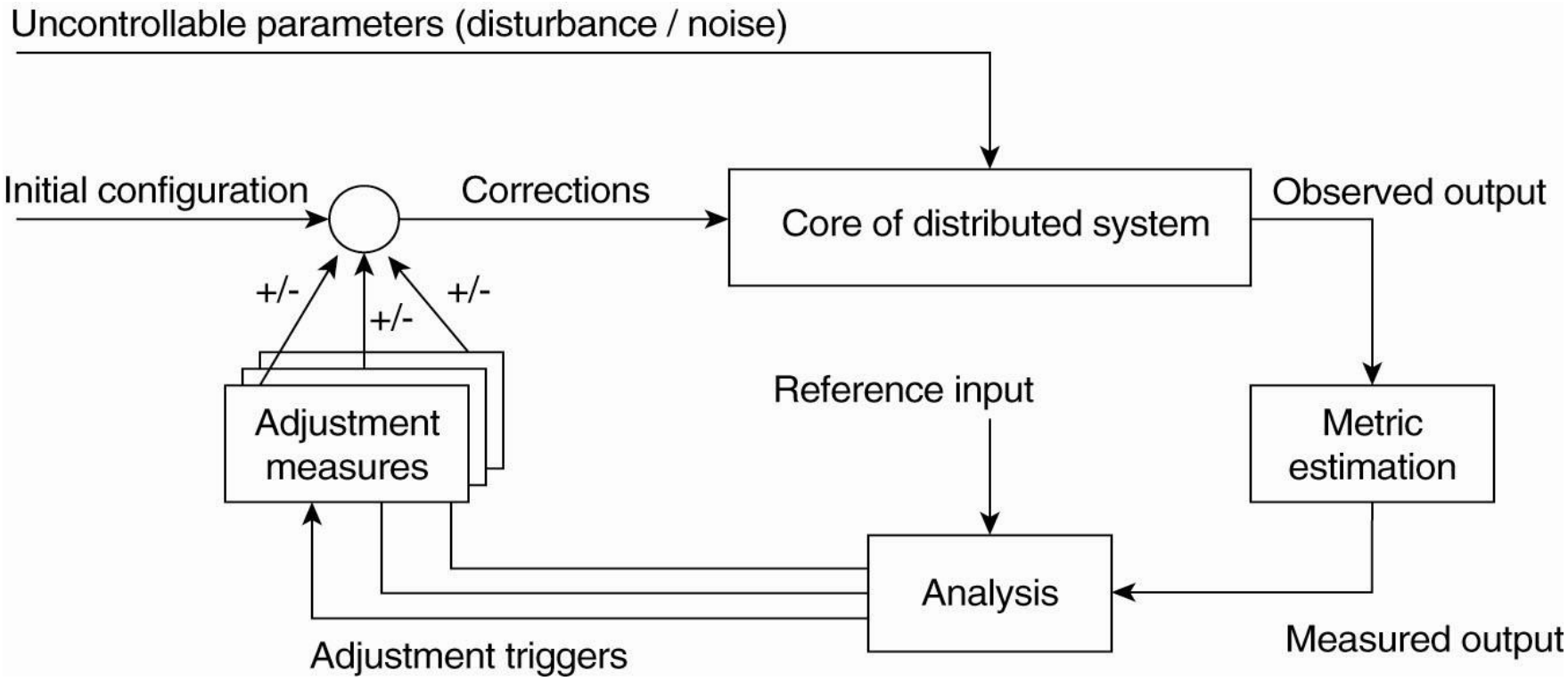


Figure 2-16. The logical organization of a feedback control system.

Example: Systems Monitoring with Astrolabe

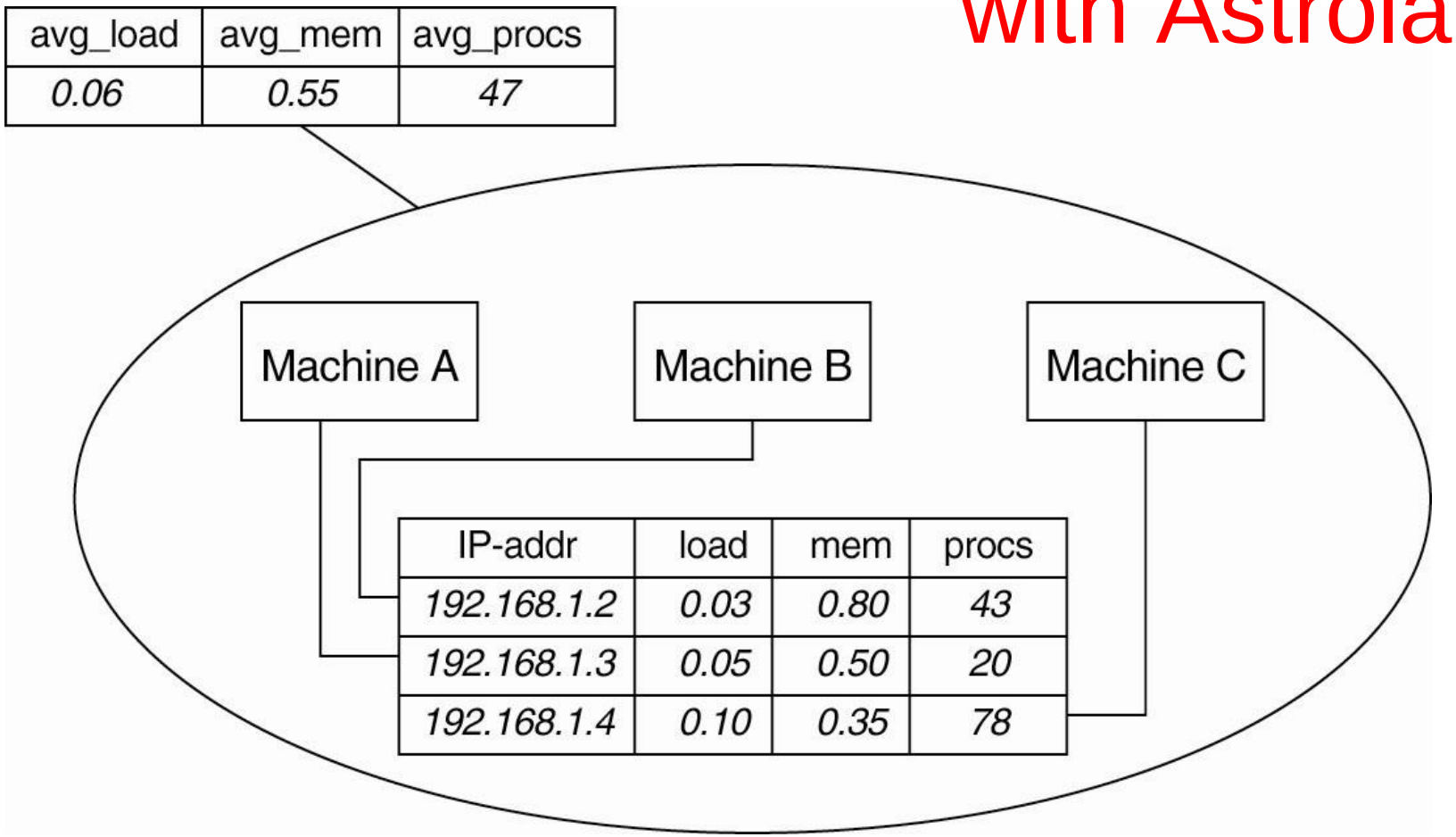


Figure 2-17. Data collection and information aggregation in Astrolabe.

Example: Differentiating Replication Strategies in Globule (1)

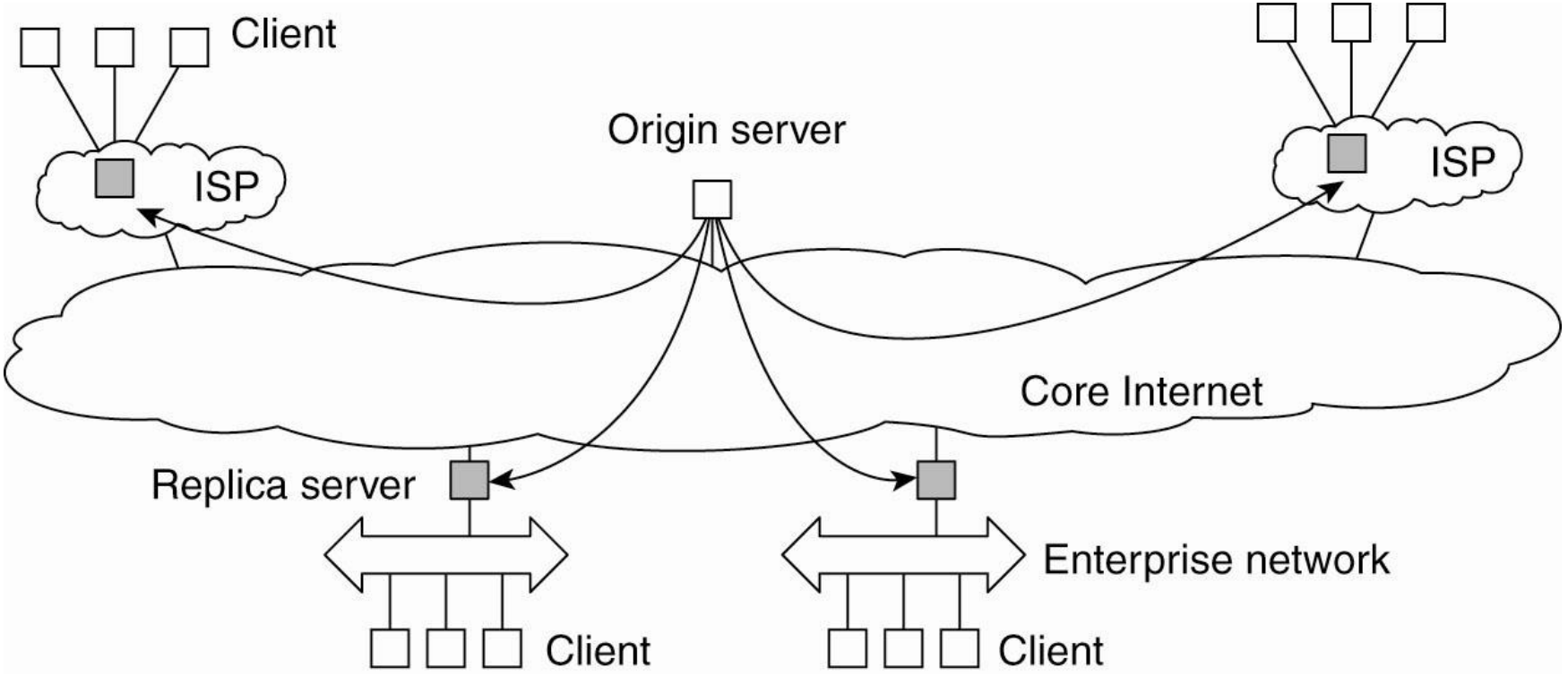


Figure 2-18. The edge-server model assumed by Globule.

Example: Differentiating Replication Strategies in Globule (2)

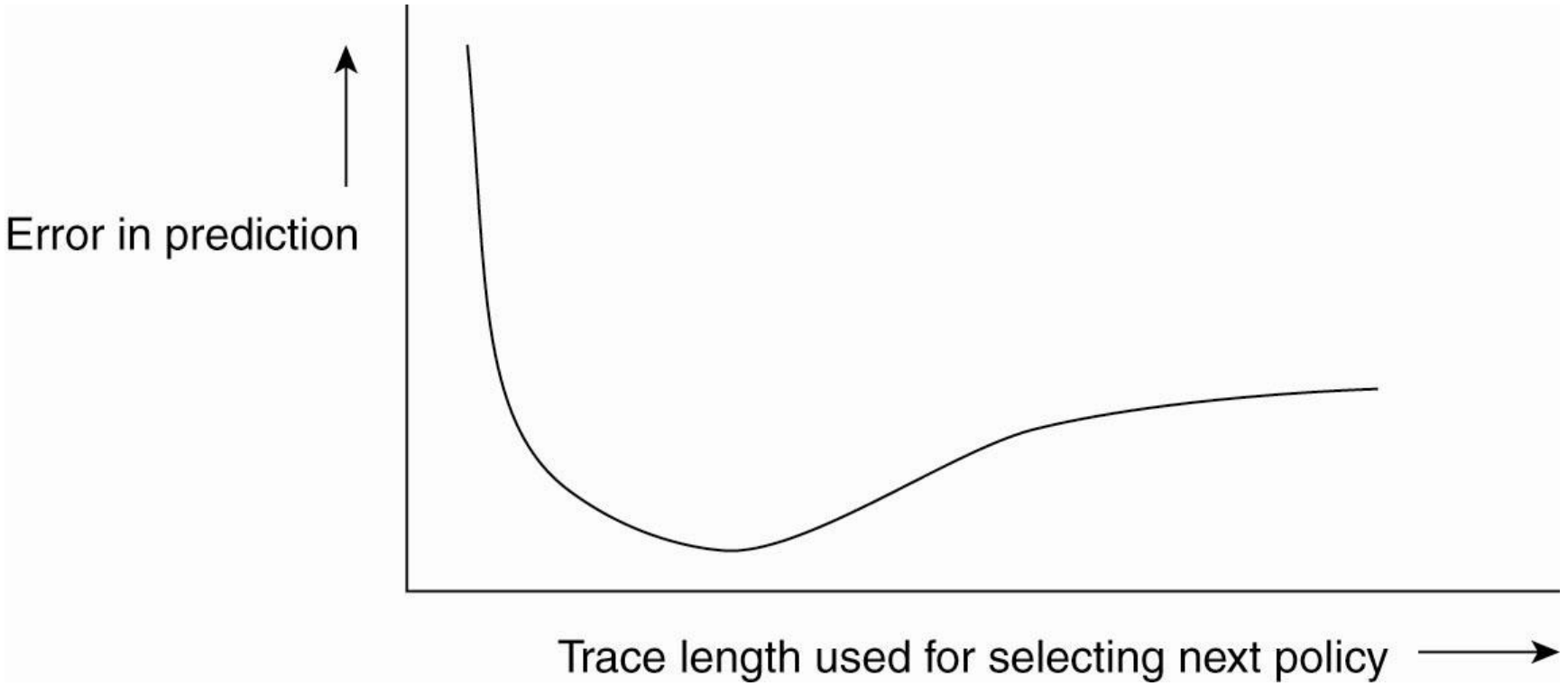


Figure 2-19. The dependency between prediction accuracy and trace length.

Example: Automatic Component Repair Management in Jade

Steps required in a repair procedure:

- Terminate every binding between a component on a nonfaulty node, and a component on the node that just failed.
- Request the node manager to start and add a new node to the domain.
- Configure the new node with exactly the same components as those on the crashed node.
- Re-establish all the bindings that were previously terminated.