

Load Distribution Patterns for Loosely-coupled Multicomputer Systems

Alexandros Karagkasidis

Institut für Technik Intelligenter Systeme (ITIS) e.V.

an der Universität der Bundeswehr München

Werner-Heisenberg-Weg 39

85577 Neubiberg, Germany

Tel. + 49 89 6004 3979

Fax. + 49 89 6004 2268

{alexandros.karagkasidis@unibw.de, karagkasidis@mail.ru}

Abstract. Load distribution is one of the main design issues that arise with respect to efficient resource management in distributed systems. Various load distribution techniques have been designed and applied in practice in many real-world systems. The goal of this paper is to present the known techniques in a pattern form. We describe six load distribution patterns for loosely-coupled multicomputer systems and relate them to each other.

Keywords: Load Distribution, Distributed Systems, Load Distribution Patterns

1 Introduction

Distributed systems began to gain popularity about 20 years ago, with new advances in technology: the development of powerful microprocessors and high-speed computer networks [Tan94, TS02].

One of the goals of distributed systems is resource sharing or, in other words, collective use of distributed resources, which are, e.g., CPUs, memory, network channels, disc space, etc. [TS02], [CDK01]. The resources of one computer, which is idle at the moment, could be used by others. For effective use, the resources in a distributed system need to be managed. Efficient resource management is one of the main design issues in distributed systems development. It influences various system characteristics. Achieving non-functional requirements, such as performance, predictability, scalability, etc., depends strongly on how resources are controlled and managed [POSA III].

A specific problem that arises with respect to resource management in distributed systems is assigning system tasks to computers (or processors). A distributed system can be seen as a collection of computers. When a system user wants to run a task or access a service provided by system, the question is which computer should handle the user request. It may be the case that some computers are overloaded, while others stay idle. Hence, to provide better performance and more efficient resource utilization, system should control the process of assigning task to computers, especially, in case of heavy and/or rapidly changing system workload. In this document, the specified problem is referred to

as *load distribution*, though other terms can be found in literature, e.g., *processor allocation*, *task assignment*, *task scheduling*, *load scheduling*, *load balancing*, or *request distribution*.

Load distribution may take various forms, depending on the type of distributed system with respect to hardware organization and software aspects. We rely on the classification scheme proposed by [TS02] and consider in this document the load distribution in context of *loosely-coupled multicomputer systems*. According to [TS02], in such systems each CPU has its own local memory and the computers are interconnected via some communication network. With respect to software, each computer runs its own operating system, called also network operating system, which differs from traditional operating system in making local services available to remote clients [TS02]. Another software-specific component used in many modern distributed systems is middleware, which lies between applications and network operating system. Middleware is intended to hide possible system heterogeneity from applications and provide additional services to them.

In this document, we present in a pattern form the known load distribution mechanisms, usually applied by system developers. We identify load distribution patterns in context of loosely-coupled multicomputer systems and show the relations between them.

The document is organized as follows: Section 2 sets up a background concerning the related work in the field of load distribution, both academic studies and practical experience. In section 3, we describe the pattern organization scheme, which relates patterns to each other. In sections 4, 5, and 6, we introduce patterns that address various aspects of load distribution problem. We describe 6 patterns. The description follows common practice of pattern representation. In section 7 we provide a summary, discussing and relating the presented patterns to each other.

2 Background

The problem of load distribution has been studied for about twenty years. In this section, we provide a brief introduction to the problem as well as an overview of the results of research work.

The research work has been conducted, primarily, in two major directions.

The first is more theoretic and addresses, in general, efficiency aspects of load distribution. With this respect, various algorithms have been proposed, analysed and compared with each other under certain conditions. They are usually referred to as task scheduling or processor allocation algorithms. However, with a great number of approaches, differing often in problem statement, terminology, assumptions, etc., it has become “difficult to analyse the relative merits of alternative schemes in a meaningful fashion” [CK88]. Attempts concerning this problem have been made. As a result, several classification schemes, or taxonomies, of algorithms of task scheduling in distributed systems have been proposed, which use various classification criteria. E.g., a general and widely accepted taxonomy is presented in [CK88].

Theoretical studies, however, were often far from practice. Regarding taxonomies, [Tan94] points out that they all are about “clear-cut theoretical issues about which one can have endless wonderful debates”, with the practical aspects remaining out of the scope. So, at the same time other researchers attempted to apply various load distribution mechanisms in practice.

Research work in this more practice-oriented direction focused initially on a model of interconnected independent workstations (referred to as *workstation model* in [Tan94]), which is typical for many organizations, e.g., for universities. Each workstation is assigned to a user or to a department. Hence, the workstations are managed and controlled independently. With respect to load distribution, the basic idea is to allow users to submit task to execution on remote workstation(s) in case of heavy load of the local workstation.

In recent years, the practical aspects of the load distribution problem have also been studied in context of new distributed technologies (such as WWW, CORBA, or DCOM), which are based on client-server paradigm. In this model, servers provide certain services, which are used by clients. The client sends to the server a service request. The server processes the request and returns the results to the client. The client-server interaction is determined by protocol. In such systems, with large volumes of load, the server side of a system is often deployed on a group of computers, or servers (referred to

as *pool of servers model* [Tan94]). Hence, the servers are managed and controlled in a centralized manner at the server site, as opposed to workstation model with independently controlled workstations. With respect to resource management and load distribution, the main goal is to efficiently distribute the client requests among the servers. For about the recent decade, a variety of solutions have been proposed in this area, especially, for Web systems. There already exist works that summarize the available information and classify the known approaches with respect to particular technologies, e.g., [CCY99], [CCCY01], [Jewell00].

Along with research studies, both theoretical and practice-oriented, load distribution techniques have successfully been applied in various real-world distributed software systems, such as web-servers, enterprise applications, clusters and grid-based systems. It means that they not only represent an object of interest for research studies, but also closely relate to actual practical issues and needs of industrial software development.

Summarizing, we may now state that gained practical experience and theoretical results could serve the developers as a solid knowledge base when developing new systems, eliminating the need to reinvent the already existing processor allocation solutions. Hence, the common load distribution techniques can be regarded, or viewed, as patterns. This document provides the corresponding representation of existing experience in a pattern form.

3 Pattern Organization Scheme

Given all this large body of information, our task was to identify, or discover, load distribution patterns. In our case, analysis of the problem area and study of existing literature, related both to theory and practical aspects, have shown that implementation of load distribution mechanisms in real-world systems involves a set of issues to be addressed. With respect to efficient resource management and load distribution, the main task to be solved is to provide the *server allocation mechanism*, or algorithm, which specifies how a system selects server where the task is to be executed. Besides, the developers must decide how to organize a system, that is, how a system structure looks like, which components are involved, how they interact with each other, how the users access the resources, how the user task is passed to the server, etc.

From this point of view, it would be possible to concentrate particularly on the problem of efficient server allocation, leaving other problems out of scope. However, our strong belief is that all the related issues should be considered in complex, which is also based on the fact that this approach dominates in the literature on practical implementation of the load distribution mechanisms. Patterns based on the comprehensive problem consideration will be more useful and helpful for system developers.

Therefore, we organize our patterns as follows.

Two basic models of loosely-coupled multicomputer systems have been mentioned above – the workstation model and the pool of servers model. We use these two models as a primary basis for pattern identification and organization. Actually speaking, they represent the general context, which determines the whole set of issues that arise when applying load distribution mechanisms in practice.

Thus, we first present a group of patterns that provide the general solutions, or approaches, addressing in complex all the issues related to the load distribution problem. These are **Cooperating Peers** pattern for the workstation model and three patterns – **Front-End Dispatcher**, **Cooperating Servers**, and **Smart Clients** – for the pool of servers model.

Then we provide patterns that are intended to solve specifically the server allocation problem, which is common for both models. These include **Static Server Allocator** and **Dynamic Server Allocator**.

Figure 1 illustrates our pattern organization scheme.

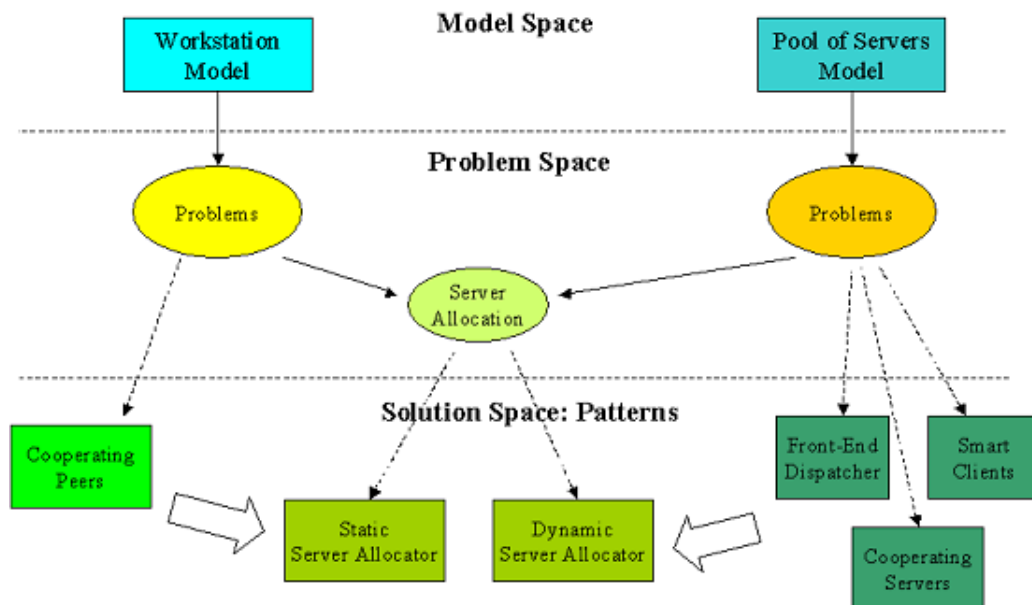


Figure 1. Pattern organization scheme.

4 Load Distribution Patterns for Workstation Model

In this section, we present a single pattern, *Cooperating Peers*, which specifies high-level load distribution mechanism for networks of workstations.

4.1 Cooperating Peers

Context.

This pattern applies to the systems that follow the workstation model of processor organization. The *workstation model* [Tan94], referred to also as *clusters of workstations*, [Tan02], consists of workstations or personal PCs connected through LANs. Such systems are typical for large organizations where each workstation is assigned to and exploited by, e.g., a single user or a department. A system is loosely-coupled, that is, the workstations are, in general, independent from each other and are controlled independently. Still, they can “interact to a limited degree where that is necessary” [Tan94]. Such interaction includes, e.g., file sharing and accessing files of remote workstations, remote login and remote task execution and is provided by the underlying operating system, which is called network operating system [Tan02].

Problem.

In general, users do not use their workstations all the time, and the workstations may stay idle for considerable amount of time. Or the workstations may not be heavily loaded, that is, their users perform some work that is not computationally intensive. At the same time, other users may need additional computational power to perform their tasks, but cannot make use of the idle and underutilized resources [LLM88], [Tan94]. This means that the resources are managed inefficiently. Hence, we need a solution that would allow the system users to exploit remote resources, i.e., the resources of other workstations. However, a set of problems arises here. In general, the solution should resolve, or balance, the following forces:

- It should be able to find idle or underutilized workstations (hence, it is necessary to monitor system state and exchange state information among workstations).
- Users should have access to remote resources, i.e., be able to run tasks remotely.
- The solution should decide among possible candidate workstations for remote task execution, and possibly the most optimal solution should be selected.
- The solution should be scalable.
- The solution should provide environment transparency for remote task execution, i.e., the task should run in the same environment, as it would have on the local workstation. This includes, e.g., file system view, environment variables. The solution should also handle the system calls made by task.
- While the task is executed remotely, the user of the remote workstation may require resources for his/her own needs. In this case, the remote task should be stopped in order to release resources for local needs. For large tasks, it would be not efficient to run them again from the beginning. Ideally, the solution should provide a checkpointing mechanism that would save the state of the task, so that it is possible to continue its execution somewhere else.
- The mechanism should imply low overhead.
- Additionally, it should be possible to monitor the running task and recover from failures.

Solution.

The workstations are in essence independent, that is, they are purposed first of all for their users and initially are not aimed for collective use. As we see, the remote task execution raises a set of issues that do not relate directly to the problem of the efficient resource management. This results in a complex solution, which is implemented as a special system service that would control the workstations in the system and provide functionality for task allocation. The service functionality usually includes registering workstations in the service, monitoring the workstation state and exchanging state information, selecting workstation for remote execution on user demand (server allocation or scheduling), submitting task for remote execution, providing (to a certain extent) environment transparency for remote task execution, passing results back to the user, and making checkpoints of task execution.

Workstations first register themselves in the service in order to be able to use remote resource when necessary and to accept remote task when possible. This makes workstations known to service. When a workstation is registered, the service monitors the workstation's state. This information is used when the service decides where to send a task for remote execution. When remote workstation is selected, the service submits the task along with required files from the sending workstation to the receiving workstation. The results are delivered back to sender. The service stops the task execution on demand and lets resume the task execution on another node, using checkpoints.

Structure.

Figure 2 provides the logical structure of the task allocation service, which involves the following participants:

- *Service user*. This is typically owner of a workstation.
- *Registry* is an element that is responsible for registering and unregistering workstations in the service on user behalf.
- *Access control*: Maintains information on the users and checks if a user is allowed to run task on a remote computer.
- *Remote task submission*. This component is requested by the user when remote resources are needed for task execution. For this purpose, the remote task submission component asks the server allocator to find remote workstation and submits task for remote execution on the selected workstation. This component also sets up execution environment on the remote workstation.

- *Server allocator* is responsible for selecting workstations for remote task execution. This component actually hides all the details related to this process, such as monitoring workstation states, gathering and exchanging the load data, which is used when making selection decisions. The Dynamic Server Allocator pattern described in section 6.3 provides server allocation solution based on system state data. Hence, it can be used to implement the server allocator component. For more details, refer to the Dynamic Server Allocator pattern in section 6.3.
- *Remote task execution, monitoring and checkpointing*: This element is used to control the remote task execution, e.g., to check its progress. It also performs checkpointing for the remotely executed task, in order to be able to restart it from the saved checkpoint, if the owner of the remote workstation requires the task execution to be stopped.

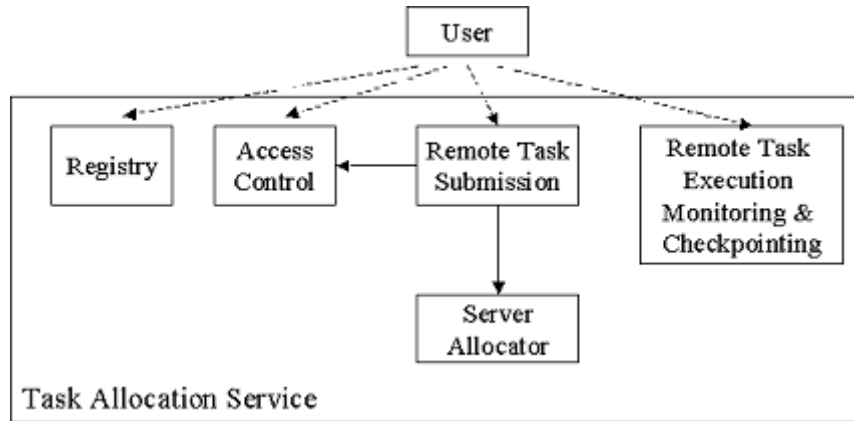


Figure 2. Structure of task allocation service

Figure 3 represents the solution as a middleware service that interconnects the workstations and is implemented over the operating system level.

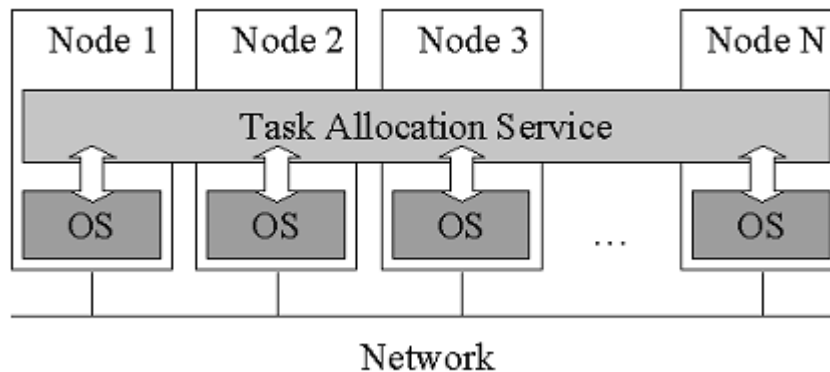


Figure 3. Task allocation service: High-level physical view.

Dynamics.

Interaction between the service user and the service includes several scenarios, such as registration, submission of the task to remote execution, and monitoring the remote execution process.

Here, we describe only the most interesting scenario, namely, submission of task to execution on remote workstation (illustrated in figure 4):

- User specifies the task, which is to be executed. This is usually an executable file, possibly supplemented with input and output files, environment variables, etc. Depending on the

service functionality, the user can also specify minimal resource demand required for task execution.

- Remote task submission component first checks access rights for this user in order to filter out in advance those workstations, where the user is not granted to execute tasks.
- After that, server allocator is requested for suitable workstations where a task could be run.
- For this purpose, the server allocator gathers state information of those workstations that have not been filtered out on the second step, selects the most appropriate one, which is then returned to the remote task submission component. The detailed description of this process is provided by the Dynamic Server Allocator pattern in section 6.3 and is not represented on the diagram.
- The remote task submission component sets up the environment on the selected remote workstation and then sends the task for remote execution.

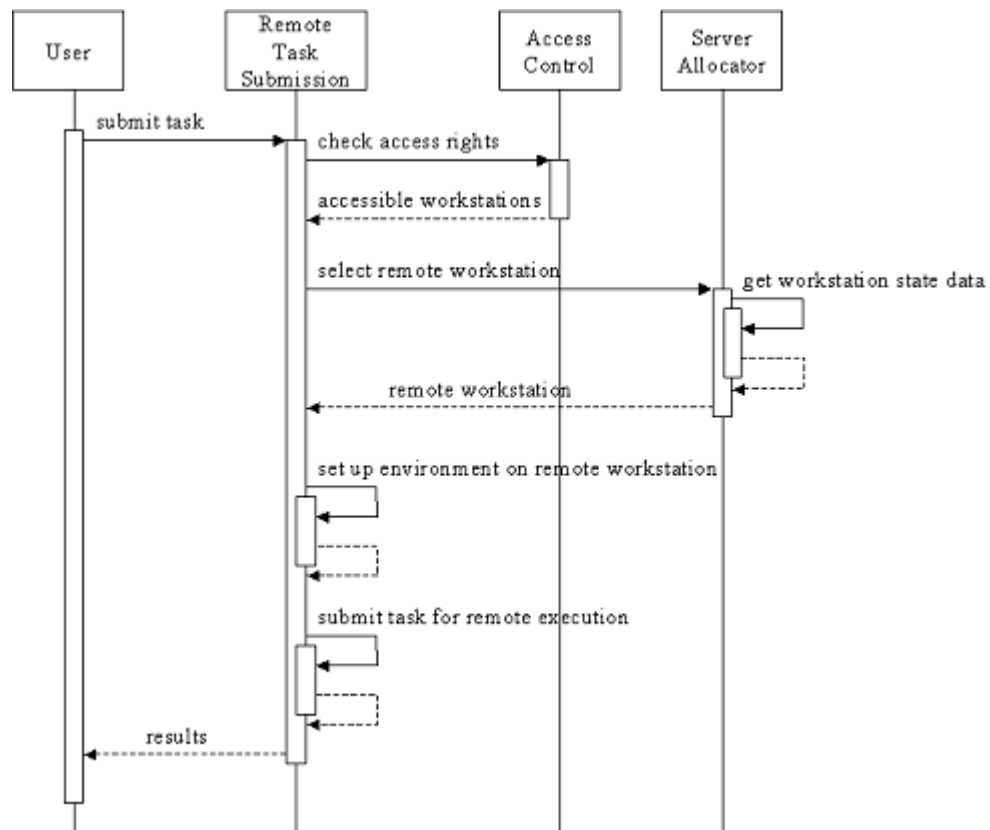


Figure 4. Submitting task to remote execution.

Implementation.

Implementation of the service involves the following aspects:

- *Implement workstation registration functionality.* In the simplest case, the network address of the workstation is provided to the service. Additionally, users could specify to the service various workstation parameters, such as CPU type, available memory, type and version of the operating system, etc.
- *Implement access control component.* The most natural and straightforward is to use the authentication/authorization mechanisms of the underlying operating systems.

- *Implement server allocator component.* Here, we refer to the “Implementation” section of the Dynamic Server Allocator pattern (in section 6.3).
- *Implement remote task submission functionality.* Implementation will involve two participants: one on the task sending workstation, the sender, and one on the task receiving workstation, the receiver. The sender prepares all the required files and data and passes them to the receiver. The receiver is responsible for setting up the execution environment, launching the remote execution of the task, and returning the results. Hence, the developers must also implement the interaction protocol between sender and receiver. The issues arise when the task uses some local settings, e.g., environment variables, or local files. In this case, the remote workstation should provide the same system view as the local one. This should be provided by service. The system calls are also a common issue in providing system transparency. But, in general, it is difficult to provide transparency for all possible cases. E.g., a task may use some files that the user is not aware about. Another problem is if a task is passed to distinct platform, with another operating system. In this case, there should exist an executable file of the task for that platform.
- *Implement remote task execution monitoring and control.* Implementation of this component will also involve two components, local and remote. Remote component will reside on the remote workstation, perform monitoring and checkpointing of the remote task execution and send corresponding data to the local counterpart, which presents the monitoring data to the user and uses the saved task state to restart it elsewhere. Implementing of monitoring will require using system calls to local task manager to obtain the execution status. Checkpointing is rather difficult because it requires finding and saving all the data related to the task execution that resides in the operating system kernel, e.g., registers, opened files, signals, etc.

Known Uses.

The first systems that exhibit Cooperating Peers pattern have been implemented about twenty years ago. Today, this pattern represents the starting point in implementation of resource management in Grid-based systems. Here, we name as examples some of the original as well as recent solutions.

- The Butler system [Nichols1987] represents a set of programs that give users access to idle workstations. The *registry* program maintains a list of currently idle workstations. The *rem* program on the client machine first contacts the *registry* to find a candidate remote workstation to use. Then the *rem* program checks the availability of the found machine by contacting the *butler* program that runs on that remote machine. If the answer is positive, the *butler* program contacts the *registry* and removes the remote workstation from the list of idle workstations. The *rem* program, in its turn, makes preparations for remote execution by setting up execution environment on the remote workstation and invoking the desired command. The *butler* program actually runs that command. When the owner of the remote machine reclaims its workstation, the *butler* program first notifies the client and then simply kills the remote task. Using the distributed file system provides the same file system view. The Butler system also directs the standard input and output streams back to the user.
- Condor system [LLM88]. The Condor system provides task scheduling, remote task execution, execution control, and automatic checkpointing (as distinct from the Butler system). Each workstation keeps the state information of its own jobs and is responsible for scheduling them; it has a local scheduler and a local task queue. One of the workstations holds also the central coordinator. The central coordinator polls periodically the workstations to see which of them are currently idle and which have tasks to be executed. Given this information, the central coordinator allocates the remote capacity of idle workstations to local schedulers on workstations that have tasks waiting. A Remote Unix (RU) facility [Litzkow87] is responsible for remote task execution. When a task is to be executed remotely, a shadow process is created locally as a surrogate of the process running on the remote workstation. When a remote process makes a system call, special library routine is invoked, which communicates with the shadow process and sends a message

containing the type of system call and marshaled arguments. RU facility also provides checkpointing: when a process receives a signal, the RU library catches it and saves the necessary data, such as registers, opened files, etc.

- Globus system. [CFKKMST98] describes architecture for resource management used in the Globus project [Globus]. This architecture involves the following components: *resource brokers*, *resource co-allocators*, *resource allocation managers*, and *information service*. They communicate with each other through requests in *resource allocation language* (RSL). Users are responsible for providing high-level RSL specifications where they describe their resource requirements. These high-level RSL specifications are then transformed by *resource brokers* into more concrete ones, where the locations of the required resources are completely specified, and passed to *co-allocator*. *Co-allocator* breaks a request involving resources at multiple sites into sub-requests and passes these sub-requests to the appropriate *resource allocation managers*. Information service is responsible for providing information about the current availability and state of the resources. This information is actually used when locating resources. In Globus system, this information service functionality is implemented in Metacomputing Directory Service (MDS) [CFFK01]. Actually, the *Cooperating Peers* pattern describes the structure and implementation of *resource allocation managers*, called also Globus Resource Allocation Managers (GRAMs). A GRAM is responsible for processing RSL specifications representing resource request; remote monitoring and management of jobs; updating MDS information about the current state of the resources it manages. The components of the GRAM are the GRAM *client* (requests resource allocation and process creation), the *gatekeeper* (responsible for authentication based on GSI - Globus Security Infrastructure), the *RSL parsing library*, the *job manager* (responsible for creating the actual processes requested by user and monitoring), and the GRAM *reporter* (provides to MDS additional information about local scheduler structure and system state, e.g., number of jobs).
- Condor-G [FTLFT01]. This system is aimed to combine inter-domain resource management protocols of the Globus Toolkit and intra-domain resource management methods of Condor. The solution actually involves the components we have described above for Condor and Globus respectively, including shadow process and system call trapping and checkpointing library of Condor system and gatekeeper, job manager of Globus Toolkit. The Condor-G *scheduler* responds to user request to submit jobs for Grid execution by allocating the resources (given available resource state information) and creating the Condor-G *grid manager*. *Grid manager* then acts as a standard GRAM client from the previous example. Special Condor-G daemon collects information about resources on the task execution site and passes it to the Condor-G *collector*, which provides this information to the Condor-G scheduler.

Consequences.

The benefits:

- Solution provides more efficient use of loosely coupled distributed resources.
- High reliability.
- High extensibility.

The liabilities:

- Very complex solution, with many issues involved. In heterogeneous systems, additional problems arise.
- Resource are not used to their full capacity.

See Also.

Static Server Allocator and Dynamic Server Allocator patterns (in section 6).

Discussion.

The complexity of the solution is caused by the fact that the resources are controlled independently and belong to independent owners that only agree upon collective use of their resources under certain conditions. That is why we have called this pattern **Cooperating Peers**: the peer owners cooperate together in order to achieve more efficient resource utilization. In other patterns that follow the available computational resources are controlled at single site, hence it is much more easier to manage them and other solutions apply in that case.

Actually, the **Cooperating Peers** pattern represents a general approach. Particular aspects of solution such as authentication issues, remote execution and management, providing transparency, e.g., handling system calls etc, should be considered separately.

Because of high complexity of the presented solution, one would likely use the existing off-the-shelf products (especially, if a system is heterogeneous and involves many computational sites, or domains, that belong to different organizations), rather than implement own service from the scratch. Typical would be to install one of existing Grid solutions, e.g., Globus Toolkit or Condor.

5 Load Distribution Patterns for Pool of Servers Model

5.1 Preface

In this section, we present three higher-level patterns - **Front-End Dispatcher**, **Cooperating Servers**, and **Smart Clients** - that apply to the systems, which follow the pool of servers model [Tan94]. This implies that, in general, the context and the problem sections of a pattern description form will be common for all three patterns. As it also turns out to be, the solutions provided by these patterns have much in common. Namely, the structures of the patterns involve similar components. However, they are organized and interact with each other in different ways. These differences let us distinguish between the patterns.

Hence, we first present aspects that are common for all three patterns and then the other parts of a pattern description, which are specific for each particular pattern.

Common Context.

The pool of servers model [Tan94] represents common context for three patterns of this section - **Front-End Dispatcher**, **Cooperating Servers**, and **Smart Clients**.

This model is actually another approach to organize a loosely-coupled multicomputer system. In contrast to the workstation model, where the users are assigned powerful personal workstations, the main computational capacities in this model are provided as a pool of powerful computers, or servers, while the users can be given simple and cheaper graphic terminals. When a user needs to execute some task, one of servers is allocated on user demand, and the task is executed on that server [Tan94]. The servers are controlled in a centralized manner. This allows better performance and resource utilization potential, as compared to the workstation model.

The systems of this type follow the classical client-server computation paradigm. Servers provide certain functionality, or services, which are accessed by clients by sending service requests to the servers. Client-server interaction is determined by protocol between the client and the server. Access to the services is possible only through this protocol.

Here, we do not make great difference between such terms as users and clients, as well as between tasks and client requests. The client may actually be a single person, or user, that wants to run his/her task, as well as a client program operated by user that sends a request to the server. From the solutions' perspective these details do not play significant role.

For clarity, however, we consider clients to be client programs, while we consider servers to be the software components that provide service(s).

Problem. Common Aspects.

As it has been mentioned above, implementing load distribution implies a set of problems to be addressed.

The main design problem in the pool of servers model is to implement server allocation mechanism, which assigns user tasks (or requests) to pool servers or, in other words, selects a server from pool to process a request. Simple solution such as letting clients select servers themselves is not suitable in most cases. First, the clients will tend to choose the “first” servers in the given server list. Second, often the distributed nature of the server side is to be hidden from the clients that should perceive the system as if it were a single server and, hence, have not direct access to servers.

In general, the tasks may differ significantly in amount and type of resources they need for execution. Some tasks, e.g., may be computationally intensive with high demand for CPU capacity and available memory, while others are more I/O intensive, performing operations with files or transferring large amount of data through network. This complicates the server allocation process with respect to the efficiency of allocation decisions. In other words, the diversity in resource requirements among user tasks makes it difficult to implement server allocation mechanism that would distribute the overall load in the most optimal way. Actually, the optimal solution can be achieved only if complete information about the resource demands of each particular task as well as the load of the pool servers is known. However, in real systems, the complete knowledge is rarely available, but some task characteristics may be known in advance. E.g., the number of requests to a web server follows certain pattern, with peaks in the middle of the day. Anyway, certain allocation mechanism must be provided, and it should be efficient as much as possible.

Another design issue related to the pool of servers model is to set up communication protocol between clients and servers, that is, how the client communicates to the server and specifies the desired service or task to be executed and how the server replies with the execution results. We put emphasis on this question, because the applied load distribution mechanism implies certain changes in client-server communication, as compared to the case with single server.

In general, the main forces to be balanced are:

- The solution should provide certain server allocation mechanism that would distribute the overall load among servers of the pool.
- This mechanism should be efficient. Two aspects of efficiency of the server allocation mechanism are to be considered. First, efficiency with respect to *load distribution quality*, i.e., the mechanism should provide efficient resource utilization and possibly prevent the situations when some servers are overloaded, while others stay idle. Second, the mechanism itself implies additional runtime overhead and use of server resources. Hence, another requirement is that this overhead is tolerable, that is, the win of applying the mechanism should exceed the implied overhead (or, one can also say, system performance characteristics should not degrade when applying the mechanism).
- The solution should be scalable, that is, it should be able to handle the increase of users or system load and consequent adding additional servers in the server pool.
- The implementation should not be complex.
- Within selected solution, developers should provide an interaction protocol between the clients and the servers (which determines how clients specify their requests, or tasks, and how servers reply with results).
- Sometimes, client interacts with server in a way when the result of the subsequent request depends on the result of the previous request. This is often referred to as an interaction within a *session*. If the requests of a session are dispatched to different servers, the overall result of the session could be incorrect. The solution should address this problem.

Solution. Common Aspects.

The load distribution mechanism provided by each pattern of this group, as we’ll show below, will involve two basic, principal (logical) components. We name the first of them as *dispatcher*. It is responsible for delivery of the client request to one of pool servers, or, in other words, this component

routes the request to the server. This component is an *intermediate* between client and server. And we name the second component as *server allocator* (similar to the **Cooperating Peers** pattern). This component is responsible for selecting among servers in the pool, or it implements the *server selection policy*. These two components cooperate with each other in order to assign a client request to a server and comprise the load distribution mechanism. The patterns that follow – **Front-End Dispatcher**, **Cooperating Servers**, and **Smart Clients** - differ in how these components (along with clients and servers) are structured and how they interact with each other.

With respect to the server allocator component, it is just very similar to that of the **Cooperating Peers** pattern. As for the **Cooperating Peers**, we do not consider this component in details when describing the patterns of this section. Again, for more information, please refer to the server allocation patterns in section 6 of this document.

Now, after having discussed the common aspects, we proceed to the particular patterns applied for the pool of servers model.

5.2 Front-End Dispatcher

Context.

See above the “Common Context” subsection on page 10.

Problem.

See above the “Problem. Common Aspects” subsection on page 11.

Solution.

The main idea of this pattern is to hide the server pool from clients, providing single access point to the system through the *front-end component*. This front-end component actually implements the load distribution mechanism at the server side. Clients communicate with the front-end component specifying their requests, or tasks. The front-end component selects one of servers in the pool, passes client request to the selected server, receives results from the server and sends the results back to the client. The servers are selected according to certain policy. To the clients, the system appears to consist of a single server. All work is done in background.

Actually, the front-end component performs the following tasks:

- Interaction with clients (receiving request from client and returning results),
- Selecting server, and
- Interaction with server (forwarding client request to the selected server and receiving form it the results of request execution).

These tasks are performed by two components – *front-end dispatcher* and *server allocator* - that comprise the front-end component.

Structure.

The Front-End Dispatcher pattern involves the following participants.

- *Server* provides certain functionality to clients. Each server resides on separate computer.
- *Client* accesses services by sending requests to the server(s) and receiving request execution results.
- *Front-end dispatcher* component is responsible for communication with clients and servers, passing the client requests to the servers and returning execution results to the clients. When a client request arrives, the front-end dispatcher requests the server allocator component for a server and forwards the client request to the selected server.
- *Server allocator* component is responsible for selecting a server and is used by the front-end dispatcher. The server allocator is implemented by one of the server allocation patterns. (See also section 6 for more details).

The structure of the Front-End Dispatcher pattern is represented in the figure 5.

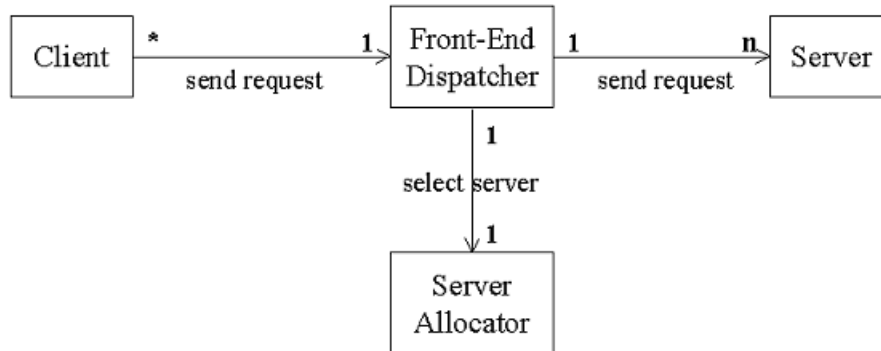


Figure 5. Front-End Dispatcher: Structure.

The UML class diagram specifies that clients interact with single front-end dispatcher component, which is connected to n servers, where n is the number of the servers in the pool. There is also single server allocator.

Actually speaking, there could also be an association between server allocator and server elements depicted on the diagram. This would correspond to the case when the server allocator uses server state information when making selection decisions (Dynamic Server Allocator pattern). However, we omit this possible association and do not specify it on the diagram. Actual relations between server allocator and server elements are presented in section 6 of this document.

With respect to load distribution mechanism, not only the logical, but also the physical structure of system is of particular interest. Hence, we provide here the deployment diagram, which shows how a system where the Front-End Dispatcher pattern is applied looks like.

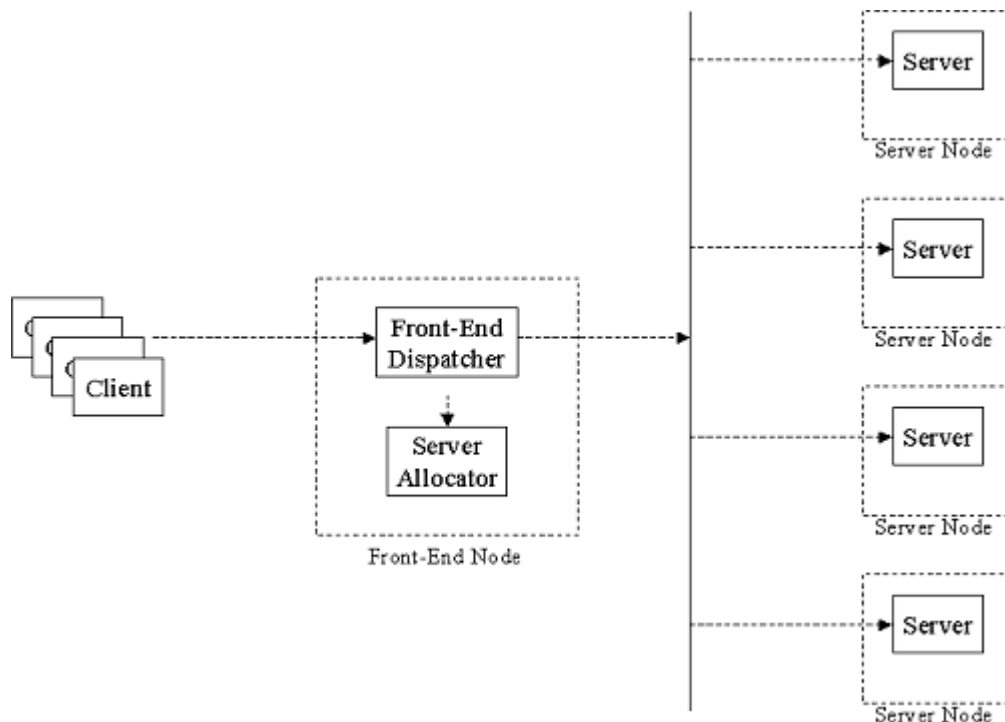


Figure 6. Front-End Dispatcher: Physical structure of a system.

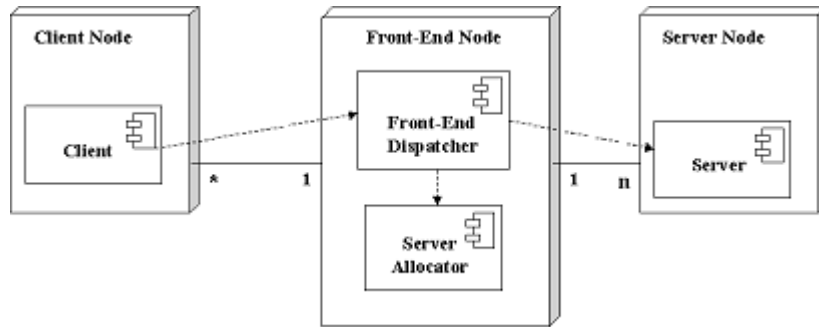


Figure 7. Front-End Dispatcher: Physical structure of a system. UML deployment diagram.

Dynamics.

In this section we describe the main scenario, which illustrates the interaction between components when processing a client request:

- The client sends the request to the front-end dispatcher.
- The front-end dispatcher needs to forward the client request to one of the servers. It asks the server allocator for a server.
- The server allocator selects one of the servers in the server pool according to its selection policy and returns the address, or location, of the selected server to the front-end dispatcher. Again, as when describing the **Cooperating Peers** pattern, we do not specify here in details the actual process of server selection that takes place in server allocator component. For detailed description, please refer to section 6.
- The front-end dispatcher then forwards the client request to the selected server.
- The server executes the request and responds with the execution results.
- The front-end dispatcher returns the results to the client.

This scenario is illustrated in the figure 8.

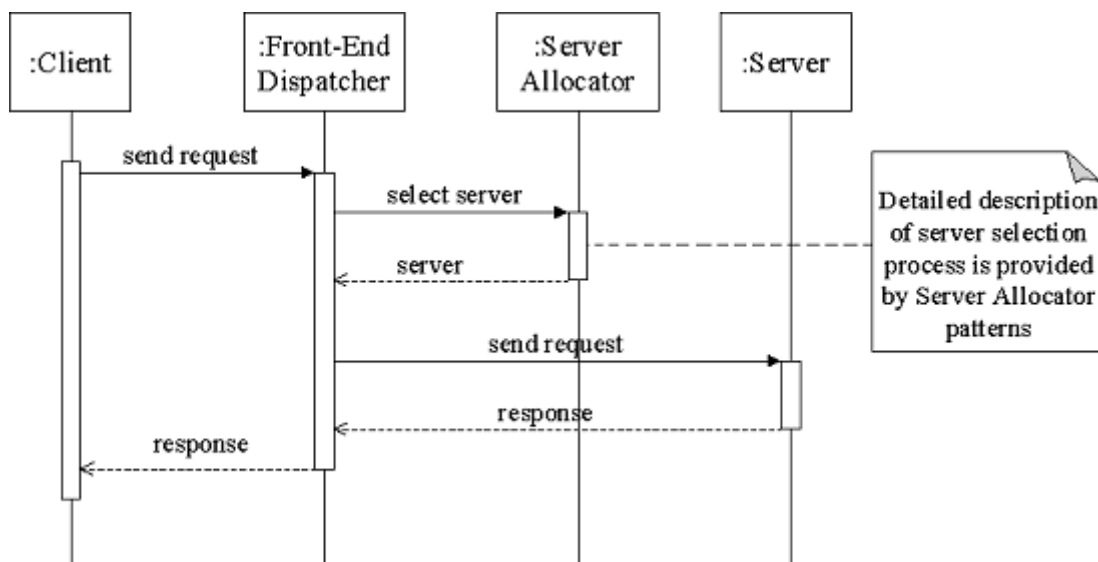


Figure 8. Front-End Dispatcher: Dynamics.

Implementation.

As compared to the **Cooperating Peers** pattern, implementation of the **Front-End Dispatcher** (as well as of other patterns of the pool of servers model) is not so complicated. It involves the following aspects:

- *Implement front-end dispatcher component.* The main issue when designing and implementing the front-end dispatcher component is the interaction protocols between the front-end dispatcher and, respectively, clients and servers. Nowadays, a distributed system is built usually upon existing distributed technology, e.g., HTTP, CORBA, or EJB. In this case, the developers will most likely make use of communication facilities provided by these technologies. However, it is also possible to use other mechanisms. E.g., when implementing request distribution mechanism in Web systems, communication between components is often designed, for better performance, at TCP/IP level, instead of HTTP (see also "Known Uses" section).
- *Implement server allocator component.* See section 6.
- *Implement session-based request distribution mechanism.* If session support is required, developers must provide additional functionality at the front-end dispatcher. This is, in general, a mechanism that maintains client-to-server mapping. With respect to particular client, the server is allocated only when the first request from this client is received. The front-end dispatcher maps the client to the allocated server. All the subsequent requests from this client are forwarded to the associated server.

Variants.

The solution described above represents the approach with indirect communication between the clients and the servers. All requests and responses pass through front-end, which therefore may become a bottleneck, depending basically on the request arrival rate and the size of responses. (E.g., when downloading high-resolution pictures or audio and video files).

Here we describe two variants of the pattern that do not require that a server pool to appear to clients as a single server.

Front-End Dispatcher with direct interaction. In this approach, clients do not send the request itself, but instead send the request for a server in the pool. The front-end in its turn do not forward the client requests to the servers. It is responsible for server allocation only: when a client sends request, the server allocator selects server from the pool and the front-end dispatcher returns the location, or address, of the selected server to the client. The client then itself establishes connection to the server and sends the request. The server returns the results directly to the client. This helps to overcome the bottleneck problem. The benefit of this approach is also that it directly provides the session-based distribution. When client obtains the server address from the front-end component, it can send all the subsequent requests to the same server. There is no need to ask for server allocation for each request.

The direct interaction variant of the **Front-End Dispatcher** pattern can be seen as an extension of the "Client-Dispatcher-Server with communication handled by clients" pattern of [POSA I], which provides server "location transparency by means of a name service". In this pattern, the dispatcher component represents an intermediate layer between clients and servers. The clients send their request directly to the servers. However, to hide the server location, the clients know only the server logical names. Hence, to establish connection to the server, the clients need the physical location of the server. For this purpose, each client asks the dispatcher to resolve server name, as the dispatcher maintains the mapping of server names to their physical locations. The dispatcher returns the physical server location to the client. The client then establishes connection to the server and sends to the server its request(s).

The next step is to allow that each server name be mapped to several physical locations and add the server allocator component that resolves the server name by selecting among available physical locations, corresponding to the specified name. Hence, the server allocator extends the standard scheme of the **Client-Dispatcher-Server** pattern with the possibility to select among several servers.

Front-End Dispatcher with semi-direct interaction. This approach is an intermediate variant between pure direct and indirect interactions. It differs from the indirect approach in that the server

returns request execution results not to the front-end component, but directly to the client. Hence, the server must establish connection to the client. This may represent certain implementation problems with respect to some distributed technologies, since the client and the server should be implemented in a way, so that they are aware of each other: the client sends request to the front-end, but is expected to be able to receive the response from the server. The server should respond directly to the client, hence the client location, or address, should be passed to the server from the front-end, which initially receives the client request.

Known Uses.

The Front-End Dispatcher is, perhaps, the most frequently used pattern, especially with respect to web systems. Here, we provide only few examples.

Known uses in Web systems:

- *DNS-Based Dispatching.* The basic idea of this approach to map the server symbolic name onto several IP addresses of the servers (instead of one address as usual) [Brisco95], [KMR95]. When a request to resolve name arrives, the authoritative DNS server for the Web site returns one of the corresponding IP addresses, depending on the implemented selection algorithm. Hence, the subsequent client requests will be distributed among servers. In [KMR95], round robin selection algorithm is used to select server (see also **Static Server Allocator** pattern in section 6.2). In [CCY99], other examples of this approach are presented, based on server or client state information. E.g., highly utilized servers can alarm the DNS server, which then excludes (temporarily) these servers from request assignments [CYD98], [SUN98], or the selection solution can be based on actual server load [Schemers95]. The main drawback of this approach is that the results of server symbolic name resolution are cached at the intermediate proxy servers as well as at the client side (such as a web browser), which implies that only a fraction of name resolution requests arrive the authoritative DNS server. This limits substantially the applicability of the approach. It is now often used in combination with other load distribution schemes (see also below, in the “Known Uses” section of the **Cooperating Servers** pattern). The DNS-based dispatching corresponds to the direct interaction variant of the pattern, where the DNS server acts as dispatcher.
- *TCP Level Routing.* Also referred to as Level-4 routing [CCY99]. In this approach, the Web servers are hidden behind the front-end dispatcher (called also Web switch or TCP router) that routes the client packets to the servers at TCP level. The clients know only the IP address of the dispatcher. When a packet arrives at dispatcher, it selects one of the servers and forwards the client packet to that server. Different mechanisms could be used by forwarding of client packets, e.g., the Network Address Translation [EF94], where the dispatcher substitutes the destination address of the packet to that of the selected server. The return packets from servers pass through dispatcher (indirect communication variant) or may be sent directly to the clients (semi-direct communication). One of the first solutions using this schema is described in [DKMT96]. A variety of other approaches can also be found in [CCY99], [CCCY01]. TCP routing is widely used in commercial products.
- *Application Level routing.* This approach is very similar to the previous one, with the exception that request routing is performed at the application level, which allows, e.g., to analyse the request content to make the server allocation decision (referred to as *content-aware load distribution*). As usual, the HTTP redirection mechanism is used to distribute client request. The front-end dispatcher receives the client request, selects server and replies to the client with redirection status code, 301 or 302 [RFC 2616]. This corresponds to the direct interaction variant of the discussed pattern.

Other examples:

- *Orbix® CORBA Naming Service Extension.* Orbix is an object request broker implemented by IONA Technologies. It provides load distribution functionality by extending CORBA Naming Service [Orbix]. Similar to the DNS-based dispatching, Orbix allows multiple

CORBA objects be registered in the Naming Service under the same name. (Of course, they must implement the same interface.) Then, when client requests the Naming Service to resolve the specified name, the Name Service returns one of the object references registered under this name.

- In [ORS01], a *Load Balancing Service* for CORBA is presented. Several object replicas reside on multiple servers and provide system service. Clients send their requests to the Load Balancer component, which acts as front-end. The Load Balancer is comprised of two components, the Replica Locator and the Load Analyser. The Replica Locator is responsible for binding clients with object replicas. When a client request arrives, this component asks the Load Analyser component to select a replica. The Load Analyser collects load data, which is used in making selection decisions. As a result, the Load Analyser returns the object reference of the selected replica. The Replica Locator then user standard CORBA LOCATION_FORWARD mechanism to redirect the client to the selected replica. Hence, the Replica Locator corresponds to the dispatcher component of the pattern structure, whereas the Load Analyser – to the server allocator. The example itself follows the direct interaction variant of the pattern. The Load Analyser collects the load data by interacting with multiple Load Monitor components that monitor the state of the replicas. With this respect, the Load Analyser and Load Monitor components implement the centralized variant of the Dynamic Server Allocator pattern (see also section 6.3).

Consequences.

The benefits:

- The multiple servers are hidden from the clients, which need to know only one entry point to the system – the front-end – that handles all incoming requests.
- Simple design: the developers need not change clients or servers. The front-end is the only new element that is added.
- The direct interaction variant provides directly session-based client-server interaction. In other variant, this feature requires additional effort to be implemented, though it is not difficult.

The liabilities:

- In general, the front-end may become a bottleneck. Hence the solution is not scalable. The two presented pattern variants – direct and semi-direct interaction – provide better solutions: they move the load implied by large responses to the servers. However, with high client request rate, the front-end dispatcher is still subject to become a bottleneck.
- The front-end dispatcher represents single point of failure. Hence, if it fails, the system is out of operation. (This problem could be solved, e.g., by adding backup-server).

See Also.

Server allocation patterns in section 6.

Actually, the solution of distributing the system load provided by the Front-End Dispatcher pattern, has already been presented in the pattern form in the previous EuroPLoP Conferences, more briefly, however. It can be found in [DL2002] as *Load Balanced Servers* pattern, and in [Sor2002], where it is called *Load Balancer*.

5.3 Cooperating Servers

Another name: Distributed Dispatcher.

Context.

See above.

Problem.

The main drawbacks of the Front-End Dispatcher pattern are that the front-end component represents a single point of failure as well as it may become a bottleneck with high request arrival rate and large volumes of responses. The two variants of the pattern – direct and semi-direct interaction – address the problem of large volume of responses through sending results directly to clients bypassing the front-end. The direct interaction variant also enables session-based server allocation, which means that the client sends the subsequent requests directly to the allocated server. This reduces the load of the front-end, because there is no need to allocate server for each request of the session, but rather only once for the whole session. However, in some cases, these solutions are not enough for sufficient efficiency and scalability. The problem is primarily inherent to the web-servers with very high request rate (or large number of concurrent users). For example, the server allocation decision may depend on the content of the request, implying that the server allocator must perform request content analysis, which can result in intolerable overhead. Hence, the Front-End Dispatcher pattern is not always a suitable solution.

Solution.

Solution provided by the Cooperating Servers pattern is intended to overcome the drawbacks of the Front-End Dispatcher.

The main idea is to make the server pool open to clients, that is, the clients are aware of multiple servers.

First, we begin with a basic scheme, which can be thought of as a starting point when discussing this pattern, the Cooperating Servers, as well as the next pattern, the Smart Clients. In this solution, the client selects a server itself and then sends the request to the selected server, which responds with the result directly back to the client. This solves the bottleneck problem inherent to the front-end dispatcher. First, the load implied by establishing connections with the clients is now distributed among the servers. Second, the responses are not passed through a single component. Also, there is no intermediate component between clients and servers, which eliminates additional communication overhead. Roughly speaking, the burden of the front-end component implied by interacting with clients and servers by forwarding client requests and returning server responses is distributed among servers. Server allocation is handled automatically by clients through manual server selection. This solution is also more fault-tolerant because if some server fails, the client can choose another one. The developers have to provide only a communication mechanism between clients and servers. As the clients are provided with a server list, they would likely select the “first” items of the list. Hence, one can say the servers are selected in a random manner, with the servers in the head of the list having greater probability to be selected. This simple approach is referred to as *mirroring* and is sufficient in certain cases.

However, in general, simple mirroring does not always resolve at least the second force of those specified above on page 11, which requires that the server allocation mechanism be efficient with respect to the load distribution quality.

The solution provided by the Cooperating Servers pattern extends this basic mirroring scheme by adding the server allocation functionality to the servers of the pool. The clients still select the servers in random order. However, each server may redirect the request to another server depending on applied server allocation mechanism. This mechanism determines, when the request is to be processed locally and when it should be sent to another server, as well as how to select among the

servers when the request is to be redirected. E.g., the server forwards the request to another server if the local load exceeds some threshold and the other server is not loaded heavily.

Hence, all servers participate in server allocation (which also explains the pattern name).

Actually, the **Cooperating Servers** pattern uses 2-level request distribution schema, with client-driven initial server selection and secondary server allocation at the server-side to “correct” the client decisions.

It must also be noted here that, besides the “manual” client-based schema, there exist other schemes of initial request distribution among pool servers. We describe some examples in the “Known Uses” section.

Structure.

The **Cooperating Servers** pattern involves the following participants, which are very similar to those of the **Front-End Dispatcher** pattern:

- *Servers* provide functionality, or services, used by clients.
- *Clients* access services provided by servers
- *Dispatcher* component is responsible for communication with clients and other servers, when forwarding the requests. It receives the client requests, passes them to the local or one of the remote servers and returns the results to the clients. When the request is to be forwarded, the dispatcher requests the server allocator to select an appropriate server. It then forwards the client request to the selected server, receives the results and passes them back to the client. This component resides at each server node.
- The *server allocator* component implements server selection mechanism. As usual, it can be implemented by server allocation patterns described in section 6.

The structure is illustrated in the Figure 9.

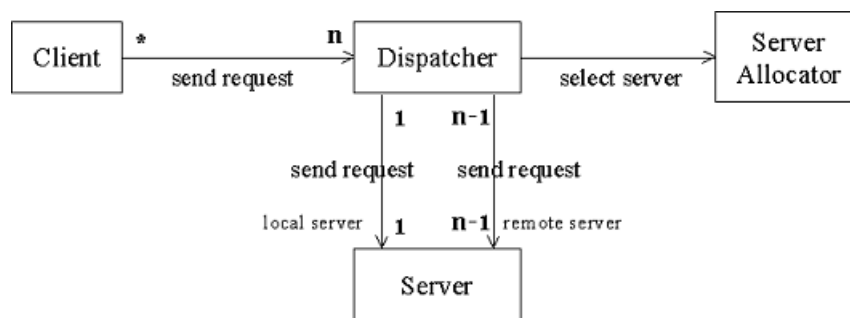


Figure 9. Cooperating Servers: Structure.

The UML class diagram specifies that clients may send requests to one of n dispatchers, where n is the number of the servers in the pool. Each dispatcher may pass the request to the local server element or to send it to one of the remote servers. The $n-1$ -to- $n-1$ association between the dispatcher and the server means that there are $n-1$ remote servers for each dispatcher, as well as there are $n-1$ (remote) dispatchers for each server. We have deliberately not specified the multiplicity of the association between the dispatcher and the server allocator elements, since it actually depends on the patten used to implement the server allocator. E.g., if the centralized variant is applied, then there will be a single server allocator element in the system, which selects the servers. This implies many-to-one association: each dispatcher asks single server allocator to select a server. If the distributed variant is used, then each server node has its own server allocator component, which is used when selecting a server. In this case, the multiplicity of association will be one-to-one: each dispatcher is associated with local server allocator (see also section 6).

As for the **Front-End Dispatcher** pattern, we provide here also the physical view of a system, which is illustrated in figure 10 (with distributed server allocator). One can see that each server node is similar to the front-end node of the **Front-End Dispatcher**.

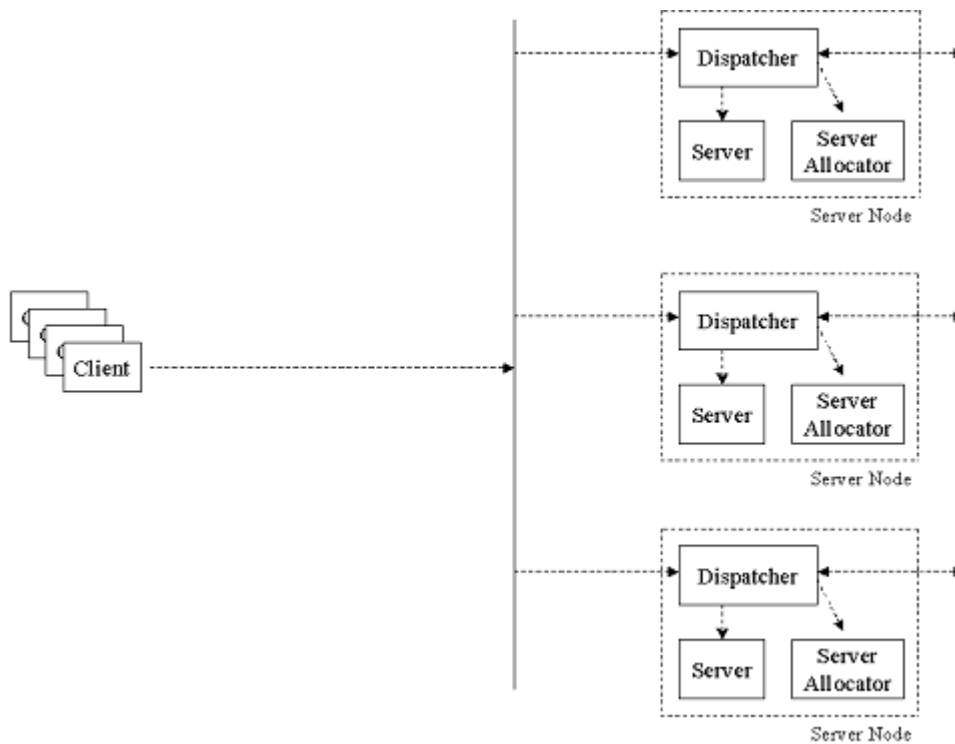


Figure 10. Cooperating Servers: Physical structure of a system

Dynamics.

In this section we describe two scenarios. The first illustrates system behaviour when client request is processed locally, while the second describes interaction between components when forwarding client request to remote server.

Scenario 1. Local request processing:

- The client sends its request.
- The dispatcher receives the client request and requests the server allocator if the request is to be processed locally.
- The server allocator determines that the local load is not high and the request may be processed on the local server.
- The dispatcher sends the request to the local server component.
- The server executes the request and returns the results.
- The dispatcher passes the results back to the client.

This scenario is illustrated in the figure 11.

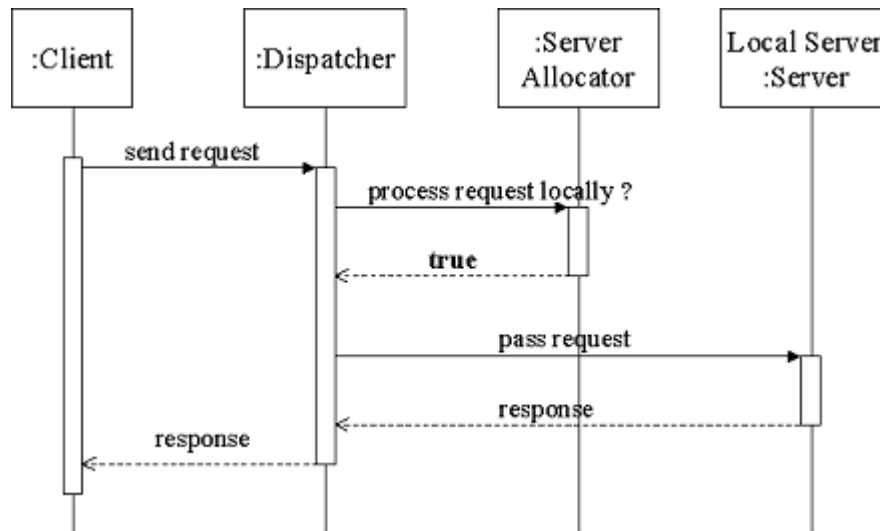


Figure 11. Cooperating Servers: Dynamics. Scenario 1.

Scenario 2. Sending client request to remote server:

- The client sends its request.
- The dispatcher receives the client request and requests the server allocator if the request is to be processed locally.
- The server allocator determines that the local load is high and the request should be processed on another server.
- The dispatcher requests the server allocator to select remote server that could process the client request.
- The server allocator selects server according to the implemented allocation policy. E.g., when dynamic server allocation is used, the server allocator obtains state information of other servers. It then analyses this information, selects a remote server and returns it to the dispatcher. (This process, however, is not shown on diagram. The detailed description is provided in section 6.) It could be the case that other servers are also overloaded. In that case, the request is to be processed locally.
- The dispatcher sends the request to that remote server.
- The remote server executes the request and returns the results to the dispatcher.
- The dispatcher passes the results back to the client.

This scenario is illustrated in the figure 12.

Implementation.

Implementation of the Cooperating Servers pattern involves, in general, the same aspects as for the Front-End Dispatcher pattern, such as, implementation of communication protocols between dispatcher and clients and servers, respectively, or implementation of the server allocator. Hence, we just refer here to the corresponding section of the Front-End Dispatcher pattern.

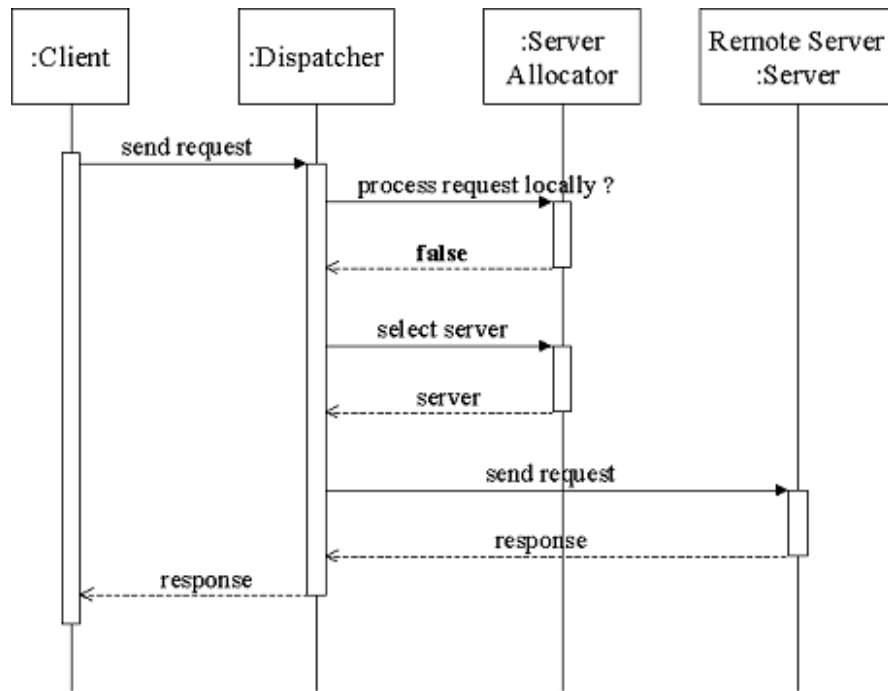


Figure 12. Cooperating Servers: Dynamics. Scenario 2.

Variants.

Redirection to remote dispatcher. In the solution described above, the dispatcher redirects client request directly to the remote server component. Another design alternative is to send the request to the remote dispatcher. In this case, the server components are not accessed directly by remote dispatchers. Each dispatcher hides its local server, receiving requests from clients and from other dispatchers. Figures 13 and 14 illustrate the structure and the dynamics of this variant, respectively.

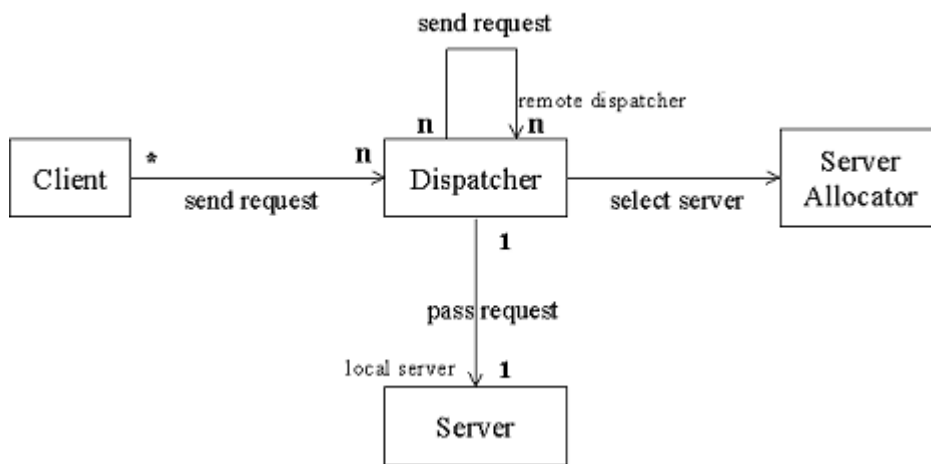


Figure 13. Cooperating Servers: Redirection to remote dispatcher. Structure.

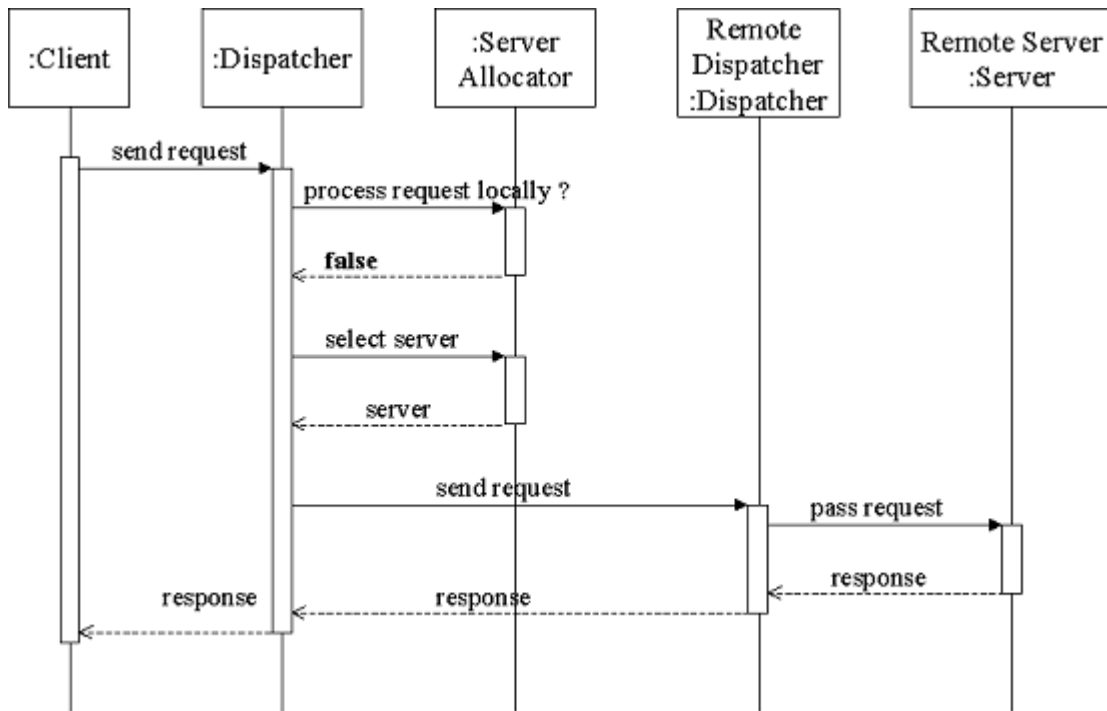


Figure 14. Cooperating Servers: Redirection to remote dispatcher. Dynamics.

The next two variants concern communication between the client, the dispatcher and the server. As for the Front-End Dispatcher, the direct and semi-direct communication schemes are possible.

Direct interaction. In this case, if the request is to be processed remotely, the dispatcher returns the location of the selected remote server. Then the client interacts with the selected server directly.

Semi-Direct interaction. In case of semi-direct interaction, the dispatcher forwards the request to another server, which then returns the results directly to the client.

Known Uses.

- [AYHI96] presents one of the first examples of the Cooperating Servers pattern in the Web. Actually, it uses two-level load distribution schema. Initially, the client requests are distributed among servers by the DNS-based solution, described above (see the “Known Uses” section of the Front-End Dispatcher). However, as it was mentioned, this solution has certain drawbacks, which limit its applicability. Therefore, the second level of request distribution is used. Each server, upon receiving a request, decides if the request is to be processed locally or to be redirected. A HTTP redirection mechanism is used. The redirection decision is based on the load state of the servers. At each node, the *loadd* component resides, collecting and exchanging this information among nodes. The *broker* component, which also resides at each node, uses the load data when making server allocation decisions.
- In [BCLM98], a *distributed packet rewriting* mechanism is described. Each server may redirect incoming requests to another server. The redirection is based on the packet rewriting through IP tunneling mechanism [IPIP]. The target, or destination, server responds directly to the client. Two approaches of the server allocation are considered. In the stateless approach, a simple hash function is used, which maps the sender IP address and port number to determine the destination server. The stateful approach is based on server load information.
- In [ASDZ00], a content-aware request distribution approach is described. It also makes use of two-level load distribution. In contrast to the first example, initially the requests are distributed among servers by front-end switch element that merely forwards the incoming

requests to the servers at the TCP/IP level. Upon receiving client request, the *distributor* component presented at each node contacts with the *dispatcher* component, which implements context-based request distribution, that is, the content of the HTTP request is analysed to determine which server should process the request. (Do not confuse the component names used in this example with those that we use in our pattern description). The difference from the previous example is that the dispatcher component (in terms of the [ASDZ00]) resides on a separate node, that is, there is a single server allocation authority in the system. (This corresponds to the centralized variant of the **Dynamic Server Allocator** pattern. See also section 6.3). The redirection is based on *TCP handoff* mechanism, which is a specific extension on the top of TCP. It is transparent to the clients, but requires modifications in the operating system on servers.

- Some interesting mechanisms are described in [CCY99a]. They also propose 2-level distribution schema, with the DNS-based first level. As opposed to the previous examples, more integrated cooperation between DNS server and the web servers is proposed, with extended DNS server functionality. E.g., DNS server may perform server load data collection, determining request allocation policy based on this information together with the client state information such as IP address, and disseminating this policy among web servers, which then use it when making redirection decisions. The redirection may be based on individual clients or client domains, which represent all clients of some network sub-domain, e.g., all clients of some organization. In another schema, the servers send alarm messages to the DNS server, indicating that they are overloaded. Upon this message, DNS server replies with the list of available servers that are not overloaded. The overloaded server uses then this list to redirect some of its requests to one of these servers.
- In [BM99], [LM01], a request distribution mechanism based on URL rewriting is described. In this solution, a Web system is deployed on a group of servers, each of which maintains a set of web documents. Generally, each web document contains hyperlinks to other documents, which may reside on other servers. The idea is to migrate the documents among servers, depending on the server load information. Document migrating requires, in its turn, rewriting the hyperlinks in those documents that refer to the migrated ones. Thus, the load is distributed among servers. Another idea is to replicate the documents, so that particular document resides at multiple servers. When a requested document contains a hyperlink to some replicated document, the server that processes the request selects among alternative replicas based on current server load information and rewrites the hyperlink accordingly.

Consequences.

The benefits:

- The problem of bottleneck inherent to the **Front-End Dispatcher** is solved.
- The solution is more scalable as compared to the **Front-End Dispatcher**.
- The solution is also more fault-tolerant. If a server crashes, the system is still in operational use. The clients can send requests to other servers.

The liabilities:

- In the **Front-End Dispatcher** pattern, the dispatching component resides on a separate node. In the **Cooperating Servers** pattern, however, this functionality is distributed among servers. This implies additional overhead at each server.
- When the server allocator resides on each node and uses policies based on the server state information, this implies additional communication between the servers due to the state information exchange, which imposes extra load on nodes and on the network.
- This solution is also more complex with respect to the implementation, as it actually requires cooperation of multiple servers, where each server is a dispatching node.
- This schema with open servers is also more vulnerable to attacks, hence is less secure.

See Also.

Patterns of server allocation in section 6.

5.4 Smart Clients

Context.

See above.

Problem.

See above.

Solution.

In the previous section, describing the solution provided by the **Cooperating Servers** pattern, we started with a basic schema called mirroring, where the clients are aware of multiple servers in the pool and select among these servers themselves. As it has been stated, mirroring represents a starting point for two patterns, the **Cooperating Servers** and the **Smart Clients**, which provide solutions to overcome the limitations and drawbacks of the mirroring scheme. The **Cooperating Servers** pattern extends this basic solution through involving the servers in load distribution.

The **Smart Client** pattern also makes use of the client-driven server selection scheme as a basis. However, instead of letting the users to select servers “manually” from the given server list, this pattern is based on idea to use special “smart” selection policies, which are aimed at providing more efficient load distribution. Hence, the responsibility of selecting server is moved completely to the clients. With respect to system structure, this means that the server allocation functionality resides now at the client side. This should be fully transparent to the users. The users should not know how a client program they operate selects a server.

With respect to the server allocation, this solution will probably use the static algorithms (see **Static Server Allocator** pattern in section 6.2). These algorithms rely on some static information when making selection decisions, i.e., random, or round robin. The dynamic mechanisms, which are usually based on server state information, such as server load, are not well suitable for client-side based server allocation. First, this would imply additional information exchange between clients and servers, which may substantially increase the network traffic and degrade the network performance, especially, with large number of the clients. Second, in some cases there are also implementation problems. E.g., if a client is a simple web-browser connecting to the web-servers. However, there exists an example, in which the server load information is used at client side to make server allocation decision. We consider here the both cases (See also the “Known Uses” section). For more information on server allocation, please refer to section 6.

Structure.

The **Smart Clients** pattern involves the same participants, as do the previous two patterns.

- *Server* provides certain functionality to clients.
- *Client* accesses services by sending requests to the server(s) and receiving request execution results.

Two additional components reside at the client side (at each client node):

- *Dispatcher* is responsible for communication with servers, when forwarding client requests. It receives the client request, then requests the *server allocator* to select server where the request is to be sent, and sends the request to the selected server. When the server replies with the results, the dispatcher passes the results back to the client.
- *Server allocator* component implements server selection algorithm.

The structure is illustrated in figure 15.

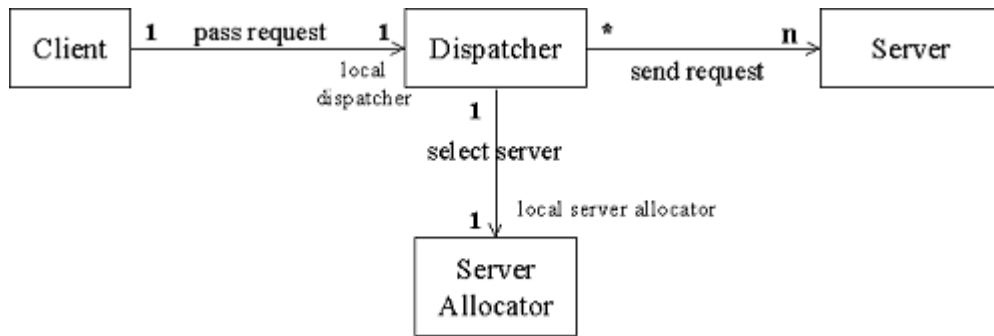


Figure 15. Smart Clients: Structure

Dynamics.

Interaction between components is very similar to that of other patterns and is illustrated in figure 16.

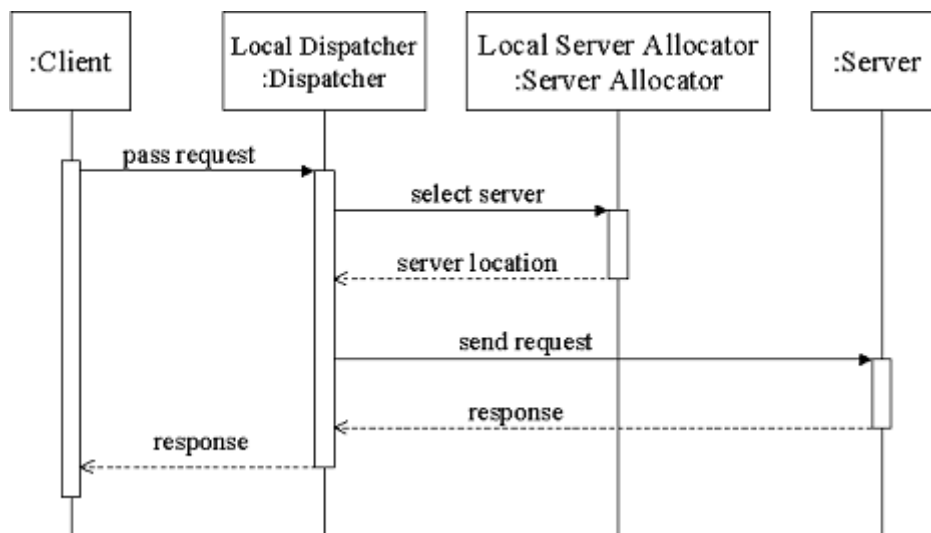


Figure 16. Smart Clients: Dynamics.

Implementation.

Here, we refer again to the corresponding section of the Front-End Dispatcher on page 15.

Known Uses.

The Smart Clients pattern is not so popular and only few examples are available.

- Smart Clients. In [YCEVAC97], a load distribution mechanism implemented at the client-side is described. It is based on Java applets, which represent executable Java code that is downloaded from the server and runs within browser on client machine. The solution uses two cooperating applets. The *client interface* applet provides the interface to the users and is responsible for accepting the user requests. It then passes the requests to the *director* applet, which encapsulates the server communication mechanism. The director applet is aware of

multiple servers. Upon receiving the request from the client interface applet, the director selects one of the servers and sends the request to the selected server. The solution proposes several server allocation techniques, e.g., random or based on server load information. In the latter case, the servers must send state updates to the director applets of the clients (see also **Dynamic Server Allocator** pattern in section 6.3).

- The **Smart Clients** pattern is applied also in CORBA and EJB. Both technologies use client stubs, which represent local proxy objects that hide from clients all the communication related issues. When calling server object method, the client actually calls method on local stub, which forwards this request to the server. Hence, the stub can be implemented so that it is aware of multiple server objects, called *replicas*, and may select among them when forwarding the client request.

Consequences.

The benefits:

- If the **Static Server Allocator** pattern is applied, there is nothing to implement at server side. Hence, there is no additional overhead at servers and on network.
- The solution is simple
- There is no request redirection, which implies better performance.
- The solution is also scalable.

The liabilities

- The use of static selection policies may not be efficient, whereas the use of the dynamic allocation algorithms implies exchange of server state information, which may cause extra overhead on network as well as on servers, especially with large number of clients.
- In general, when new servers are added, the clients need to be updated in order to become aware of new servers.

See Also.

Server Allocation patterns in section 6.

6 Server Allocation Patterns

6.1 Preface

In this section we present two patterns – **Static Server Allocator** and **Dynamic Server Allocator** - that address particularly the problem of server allocation, i.e., they specify a mechanism, which is used to select a server (or a computer) where task (or request) is to be executed. As it was mentioned, the server selection mechanism is integral part of all the solutions provided by patterns of previous two sections, where we deliberately omitted its detailed consideration. On the structural diagrams, we used the server allocator component to represent this mechanism. The patterns of this section describe in details the design aspects of this component. They can be classified as low-level design patterns (according to [POSA I] classification scheme), since they provide solutions for particular system element, the server allocator.

Actually, these patterns share, in general, the same context and solve the same problems. Hence, as in the previous section, we first consider the common aspects of the patterns and then the specifics of each pattern in separate subsections.

Common Context.

The server allocation patterns apply in the context of any of the four higher-level patterns presented above (Cooperating Peers, Front-End Dispatcher, Cooperating Servers, or Smart Clients) and specify the server selection algorithm.

Problem. Common Aspects.

The basic problem to be solved is to provide the server selection mechanism. This mechanism specifies which computer of a distributed system should handle the user task or request. In previous sections, when describing the problems related to both models (workstations and pool of servers), we have specified, among others, those forces that relate directly to this problem (on the pages 5 and 11, respectively). As one can see, these forces differ slightly from each other for the two models. This could be explained by different nature of the models, most of all because of different nature of resource control and management mechanisms. Summarizing, we can state here that the server allocation mechanism should balance the following forces:

- The mechanism should be efficient with respect to the *load distribution quality*, i.e., it should provide possibly equal load of processors and efficient resource utilization, as well as prevent the situations when some servers are overloaded while others stay idle.
- Since the mechanism implies additional runtime overhead and consumption of resources, the requirement is that the overhead be tolerable, that is, application of mechanism should not result in system performance degradation.
- The provided mechanism should be scalable, that is, it should be able to handle the increase of users or system load and consequent extension of the system with new servers.

This order, in which the forces are specified, assumes the increasing quality of the server allocation mechanism. In other words, resolving the first force is prerequisite: we must distribute system load somehow. Resolving the second force means that our mechanism not only distributes the load, but also makes it with low overhead. Depending on system performance requirements, the second force may not be present in all cases, that is, it is enough to resolve the first force only. Yet resolving the third force, along with the first two ones, makes the mechanism a perfect one: it is efficient and it will remain efficient with the growing system size, i.e., number of servers in the pool.

Actually, the forces are controversial. E.g., often to improve the load distribution quality, additional information of the servers' states is required, which implies the necessity to collect and transmit this information, which, in turn, implies additional overhead. This is an example of classical trade-off between quality and price we pay for this quality. And the decision will highly depend on which of the specified forces prevails.

6.2 Static Server Allocator

Context.

See above.

Problem.

When designing a server allocation mechanism, high quality of load distribution is, of course, desirable. However, often in many systems, this is not very important, e.g., when performance requirements are not strict. Balanced long-term load distribution is primarily of interest, whereas an imbalance in load of servers is acceptable for short periods of system work. Another problem might be that information required for more efficient load distribution is not available or implies great extra overhead to obtain, which is not tolerable. E.g., if one decides to use server state information in server selection mechanism for the **Smart Clients** pattern, this information is to be sent by servers to all clients. With large number of clients, this may impose heavy load on servers and highly increase the network traffic and eventually result in performance degradation.

Solution.

The idea of **Static Server Allocator** pattern is to provide simple server allocation mechanism, such as, e.g., round robin. It may also be based on some static, predefined information, which contains additional instructions for server allocation algorithm. E.g., in web servers, such instructions may prescribe to assign all dynamic content requests to a specified server. In other words, actual system state information, such as load of particular servers, is not used. As so, this solution cannot guarantee, in general, sufficiently high quality of load distribution. On the other hand, using simple algorithms implies relatively low overhead, perfectly resolving the second force. Hence, the philosophy of the **Static Server Allocator** pattern is “low quality for low price”. With this respect, in many cases, it would be the most appropriate solution. It is also important to note here that there are situations, where such simple approach provides rather efficient load distribution.

Structure.

Due to simplicity, the logical structure of the solution involves usually single component, the server allocator, along with some initialization information, such as server locations, or addresses. This may also include parameterization data specific to the implemented algorithm.

We would like to discuss here some aspects of the **Static Server Allocator** in context of patterns, presented in the previous section.

The described solution applies directly “as is” in the context of the **Front-End Dispatcher** and the **Smart Client** patterns. In both cases, server allocator is used by dispatcher component to obtain a server when assigning each task or request (see also the logical and deployment diagrams for these two patterns in the corresponding sections).

In the case of the **Cooperating Servers** pattern, as we have described above, each server node may redirect task or request to another server node, which is selected by the server allocator. With this respect, there are two possibilities. First, each server node may have its own local server allocator, which then is used by the corresponding local dispatcher component. This scheme is called distributed. It is represented in figure 16 in the **Cooperating Servers** pattern section. The second possible option is to have only one server allocator used by all the dispatchers, which corresponds to the centralized scheme. The server allocator may reside on a separate node or on one of server nodes. Examples of these two approaches can be found in the “Known Uses” section of the **Cooperating Servers** pattern. URL Rewriting mechanism of [BM99] represents an example of the distributed **Static Server Allocator**. Solution described in [ASDZ00] is an example of the centralized **Static Server Allocator**.



Figure 16. Static Server Allocator: Structure

Dynamics.

The dynamic is also very simple. It does not involve any interaction between components. The server allocator provides just one method, used by other components (such as dispatcher) to obtain server location. This simple scenario was illustrated above as an integral part of other scenarios.

Implementation.

The developers need only to implement the corresponding selection algorithm as well as the initialization, if necessary. Here we describe some of these algorithms that are frequently mentioned in literature and widely used in practice:

- *Round Robin*: servers are organized in a circular list with a pointer to the last selected server. When a request arrives, just the next server in the list is selected.
- *Random*. The servers are selected randomly with equal hit probability for each server. In the extended schema, the servers are assigned different probabilities according to their processing power.
- *Schedule-based*. There exist systems, where complete information on system tasks is available. That is, task arrival times and requirements on resources are known in advance. In this case, a schedule could be computed, which assigns the tasks to the servers in the most optimal or sub-optimal way. The server allocator is initialized with the schedule, which is computed in advance. From implementation point of view, it should be mentioned that is necessary to compute the schedule. This is well-known optimization problem, with many existing algorithms that are out of the scope of this document.
- *Mapping*. This is a general approach, which is based on idea to map some client- or request-specific information onto a set of servers. This information is used to divide client requests into classes according to certain criterion, or key, and then map each class onto one of the servers. Hence, the request stream is divided into sub-streams, where each of the resulting sub-streams is handled only by one of the servers. For such mapping, one can use, e.g., client IP-address, or some client identifier, such as HTTP cookie, or request content. In request-based mapping, different servers process requests of different types or provide different services. E.g., one Web-server processes only the static content requests, while the other only the requests with dynamically generated content. Or one server may provide mail service, while another search service. In data-centric system it is possible to distribute system data among multiple servers and distribute requests according to requested data. In context of the **Smart Clients**, a simple schema can be used, which binds each client to particular server in advance. In this case, a client is initialized with one of servers and uses this server every time.

Actually, these algorithms could also be seen as *variants* of the **Static Server Allocator** pattern.

Known Uses.

Static algorithms are very popular. Many of the solution described in the previous sections apply the **Static Server Allocator** pattern. Here, we just name few examples.

Round Robin is a widely used schema. It is applied, e.g., in DNS-based dispatching, in [DKMT96]. Mapping-based approach is used, e.g., in [ASDZ00] or in [BM99]. Service-based mapping is applied in many popular Web-portals, such as Yahoo, which provide separate servers for different services, e.g., mail, search, personal user web pages, etc.

Consequences.

The benefits:

- Simple implementation. The developers have to implement only the server selection algorithm.
- Low run-time overhead.

The liabilities:

- This approach does not guarantee high quality of load distribution and efficient resource utilization.

6.3 Dynamic Server Allocator

Context.

See above.

Problem.

Although the Static Server Allocator is very simple in implementation and implies low overhead, the provided load distribution quality is often not high, especially in the systems with heavy and/or rapidly changing workload, which precludes achieving system performance requirements.

Solition.

Use Dynamic Server Allocator to provide high quality of load distribution among servers.

In this pattern, the allocation decision is based on the current (hence, dynamic) state of the system. The system state information is collected on each server. Given this information, a value called *load index* is computed. The load index is intended to reflect the actual load of each particular server. When making allocation decision, the server with the lowest index is selected. This allows more sophisticated and more efficient assignment of the user tasks or requests and improves system performance characteristics.

The main issue to be solved is to determine what information should be used to compute the load index. Usually, this involves CPU utilization, available memory, opened network or database connections, number of requests in queue etc. The task of the developer is to choose load index that would correspond to the nature of application. E.g., if a client task is computationally intensive, the load index should be based on such parameters as CPU utilization and, possibly, available memory, whereas the network state would be the most appropriate load index for applications with large network data exchange.

Structure.

As compared to the static allocation, the structure of the Dynamic Server Allocator is a little bit more complex. It involves the following components:

- *Server*. In general, server is responsible for processing client tasks or requests. In context of this pattern, the server’s state is of main interest. The state information may relate to hardware components (e.g., CPU utilization) or to software components (e.g., number of client requests in queue). With this respect, the server can be viewed as either hardware or software server, or the both aspects can be considered in complex.
- *Load monitor* is responsible for collecting server state information and passing this information to the server allocator. Hence, it is presented on each server node.
- *Server allocator* obtains server state data from load monitors and make selection decisions based on this data. Interaction between these two components is discussed in the “Implementation” section.

Figure 17 illustrates the structure.

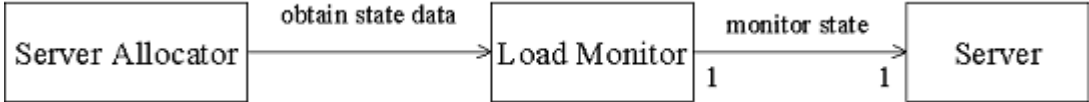


Figure 17. Dynamic Server Allocator: Structure.

On the diagram, we have deliberately omitted the cardinality of the association between the server allocator and the load monitor components. There exist several variants, which correspond to different

cases with respect to the deployment of these components on physical nodes. Actually speaking, these variants arise when applying the Dynamic Server Allocator in context of higher-level patterns described previous sections. This is discussed in the “Variants” section.

Dynamics.

Here we provide a typical scenario inherent to the this pattern:

- When a server is to be selected, the selection method of the server allocator is called.
- The server allocator queries the state information of the servers from all load monitor components.
- Each load monitor component obtains the necessary information on the server state and returns it to the server allocator.
- The server allocator computes the load indices of the servers, selects the least loaded server and returns it to the requesting component (e.g., the dispatcher component).

Figure 18 illustrates this scenario.

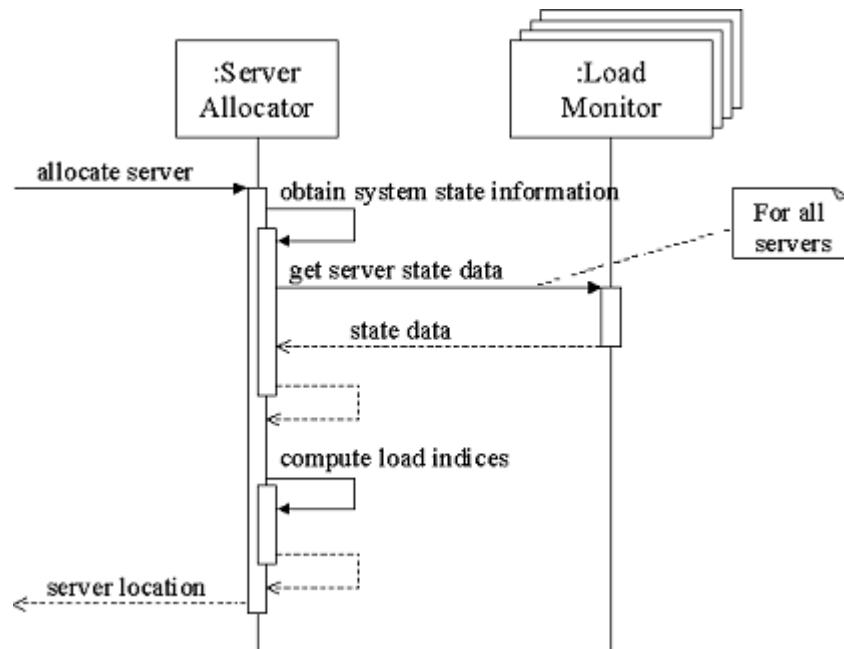


Figure 18. Dynamic Server Allocator: Dynamics.

Implementation.

- *Define load index.* The developers must determine, what system state information will be used to compute the load index and provide the corresponding algorithm.
- *Implement load monitoring functionality.* This is possibly the most difficult task for the developers, if the required state information is hardware-level. This may require using services or system calls provided by operating system or special tools.
- *Implement interaction mechanism between server allocator and the load monitors.* Here exist several options, depending on how the state information is passed. In *pull model*, the server allocator initiates the interaction and obtains the information through some interface, provided by load monitor. This approach was described in the “Dynamics” section. The server allocator may perform this periodically or upon each request. In *push model*, the server allocator provides some callback interface, which is used by load monitor

components to send state information. This can also be done periodically or when certain condition is met, e.g., when deviation in server state from the last value sent to the server allocator is greater than some threshold value.

Variants.

Here we discuss the Dynamic Server Allocator in context of the patterns presented above.

In case of the Front-End Dispatcher, there is a single node that distributes client tasks among servers. This implies single server allocator, which resides on the front-end node and obtains state information from load monitors. Hence, there is 1-to-n association between the server allocator and the load monitor.

For the Smart Clients pattern, the server allocator resides on each client node and receives information from all load monitors. The cardinality of the association in this case is “many-to-n”. (We note here again that the Dynamic Server Allocator is not inherent to the Smart Clients pattern, though possible).

In case of Cooperating Peers and Cooperating Servers, there exist two possibilities, namely, centralized and distributed schemes. In the centralized approach, there is single server allocator component, which resides on one of nodes and collects state data (hence, 1-to-n association). Another possibility is to have local server allocator on each node, which corresponds to the distributed scheme. In this case, each local server allocator obtains state data from all load monitors, which means n-to-n association between these components.

Known Uses.

Some examples of the Dynamic Server Allocator can be found in previous section, e.g.:

- In [LLM88] and [ORS01] the centralized approach is used.
- In [CFKKMST98], a GRAM *reporter* component runs on each node and monitors the systems state. This information is sent to MDS service, which provides it to resource allocators (see also the “Known Uses” section of Cooperating Peers pattern).
- Solution described in [SWEB96] is an example of the distributed scheme, where server allocator (called *broker*) and load monitor (called *loadd daemon*) reside on each server node. The *loadd* daemon updates periodically CPU, network and disk load information.
- [CCY99a] presents an example of push model of server state exchange. As it has been described above, the dispatching is performed by DNS-server, which maintains a list of available servers. When some server becomes overloaded, it sends an alarm message to the DNS-server.

Consequences.

The benefits:

- Better quality of load distribution, as compared to the static approach.

The liabilities:

- Obtaining and exchanging system state information may impose extra overhead. E.g., with high client request rate, it would be too expensive to obtain state data for each request and use better periodical updates.

7 Conclusion

In this document, we have presented patterns concerning the load distribution problem in context of loosely-coupled multicomputer systems. We first described four higher-level patterns that provide solutions addressing in complex all the issues related to the specified problem. *Cooperating Peers* pattern is proposed for workstation model. Three other patterns – *Front-End Dispatcher*, *Cooperating Servers*, and *Smart Clients* – apply to systems that follow the pool of servers model. These patterns have certain in common. For example, *Front-End Dispatcher* and *Cooperating Servers* provide server-side solutions, as opposed to *Smart Clients*. For *Cooperating Servers* and *Smart Clients*, servers of the pool are not hidden from clients, as in *Front-End Dispatcher*. In *Cooperating Peers* and *Cooperating Servers*, the constituent computers cooperate with each other to distribute system load. The difference is that in the former case workstations are controlled independently, while in the latter the servers are controlled centralized.

With respect to software development, these four patterns relate to the system architecture, to the higher-level system design, since they determine to a great extent the overall system structure. Hence, according to the classification schema proposed in [POSA I], they can be viewed as *architectural patterns*.

Then we presented *Static Server Allocator* and *Dynamic Server Allocator* patterns, which specify solutions particularly for server allocation, or selection, problem and are used by each of the higher-level patterns. In this respect, according to [POSA I], we can say that there exist the *refinement* relationship between the two groups of patterns. Indeed, each of the four higher-level patterns involves the server allocator components, which can be implemented, or refined, by one of server allocation patterns.

Hence, we can classify *Static Server Allocator* and *Dynamic Server Allocator* as *design level patterns*, as they specify solution for a particular system component [POSA I].

Concluding, we can state that the presented patterns together with their relationships provide a basis to create a pattern system for load distribution. What are missing are the guidelines that would help the user of the pattern system to select among possible alternatives. Another task would be to establish relationships to other known patterns, proposed for distributed systems, e.g., patterns for fault-tolerance (as load distribution and fault-tolerance are often used hand in hand). We consider this as our future work.

References

- [ASDZ00] M. Aron, D. Sanders, P. Druschel, W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In Proceedings of the USENIX, 2000.
- [AYHI96] D. Anderson, T. Yang, V. Holmedahl, O. H. Ibarra. SWEB: Towards s Scalable World Wide Web Server on Multicomputers.
- [BCLM98] A. Bestavros, M. Crovella, Jun Liu, D. Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures, 1998.
- [BM99] S. M. Baker, B. Moon. Distributed Cooperative Web Servers. 1999.
- [Brisco95] T. Brisco. DNS Support for Load Balancing. RFC 1794. 1995.
- [CCCY01] V. Cardellini, E. Casalicchio, M. Colajanni, P. S. Yu. The State of the Art in Locally Distributed Web-server Systems. IBM Research Report. 2001.

- [CCY99] V. Cardellini, M. Colajanni, P. S. Yu. Dynamic Load Balancing on Web-server Systems. IEEE Internet Computing, vol. 3, no. 3, pp 28-39, May-June 1999.
- [CCY99a] V. Cardellini, M. Colajanni, P. S. Yu. Redirection Algorithms for Load Sharing in Distributed Web-server Systems. Published in the Proceedings of IEEE 19th Int. Conf. on Distributed Computing Systems (ICDCS'99), 1999.
- [CDK01] G. Coulouris, J. Dollimore, T. Kindberg. Distributed Systems: Concepts and Design. Addison Wesley, 2000.
- [CYD98] M. Colajanni, P. S. Yu, D. M. Dias. Analysis of Task Assignment Policies in Scalable Distributed Web-server Systems. Published in IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 6, June 1998.
- [CFFK01] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman. Grid Information Services for Distributed Resource Sharing. Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, 2001.
- [CFKKMST98] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S Tuecke. A Resource Management Architecture for Metacomputing Systems. Lecture Notes in Computer Science, vol. 1459, 1998.
- [CK88] Thomas L. Casavant, Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. 1988.
- [DKMT96] D. M. Dias, W. Kish, R. Mukherjee, R. Tewari. A Scalable and Highly Available Web Server. In Proceedings of the IEEE Computer Society International Conference (COMPCON), 1996.
- [DL2002] Paul Dyson, Andy Longshaw. Patterns for High-Availability Internet Systems. EuroPLoP 2002.
- [EF94] K.Egevang, P. Francis. The IP Network Address Translation (NAT). RFC 1631. 1994.
- [FTLFT01] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing, 2001.
- [Globus] Globus Project. www.globus.org
- [IPIP] C. Perkins. IP Encapsulation within IP. RFC 2003.
- [Jewell00] T. Jewell. EJB 2 Clustering with Application Servers. 2000. www.onjava.com/pub/a/onjava/2000/12/15/ejb_clustering.html.
- [KMR95] Thomas T. Kwan, Robert E. McGrath, Daniel A. Reed. NCSA's World Wide Web Server: Design and Performance. 1995.
- [Litzkow87] M. J. Litzkow. Remote Unix. Turning Idle Workstations into Cycle Servers. In USENIX Summer Conference, pages 381-384, 1987.

- [LLM88] M. J. Litzkow, M. Livny, M.W. Mutka. Condor - A Hunter of Idle Workstations. Proceedings of the 8th International Conference of Distributed Computing Systems, June 1988.
- [LM01] Q. Li, B. Moon. Distributed Cooperative Apache Web Server. Proceedings of the 10th International WWW Conference, Hong Kong, 2001.
- [Nichols87] David A. Nichols. Using Idle Workstations in a Shared Computing Environment. In Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, pages 5-12. ACM, 1987.
- [Orbix] IONA Technologies. Orbix. Programmer's Guide, C++, version 6.2. http://www.iona.com/support/docs/orbix/mainframe/6.2/pguide_cpp/, 2004.
- [ORS01] O. Othman, C. O'Ryan, D. C. Schmidt. The Design of an Adaptive CORBA Load Balancing Service. IEEE Distributed Systems Online, vol. 2, 2001.
- [POSA I] F. Buschmann, R. Meunier, H. Rohnert, P. Somerlad, M. Stal. Pattern-Oriented Software Architecture. A System of Patterns. John Wiley and Sons, 1996.
- [POSA III] M. Kircher, P. Jain. Pattern-Oriented Software Architecture: Patterns for Resource Management. John Wiley and Sons, 2004.
- [RFC 2616] R. T. Fielding, J. Gettis, J. C. Mogul, H. F. Frystyk, L. Masinter, P. J. Leach, T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616. 1999.
- [Schemers95] R. J. Schemers. Ibmname: A Load Balancing Name Server in Perl. In LISA IX. 1995.
- [Sor2002] Kristian Elof Sorensen. Session Patterns. EuroPLoP 2002.
- [SUN98] A. Singhai, S.-B. Lim, S. R. Radia. The SunSCALR Framework for Internet Services. Proceedings of the 28th IEEE Symposium on Fault Tolerant Computing Systems, 1998.
- [Tan94] Andrew S. Tannenbaum. Distributed Operating Systems. Prentice Hall, 1994.
- [TS02] Andrew S. Tannenbaum, Maarten van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, 2002.
- [YCEVAC97] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, D. Culler. Using Smart Clients to Build Scalable Services. Proceedings of Usenix, 1997.