A System of Patterns for Concurrent Request Processing Servers

Bernhard Gröne, Peter Tabeling

Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany

`{bernhard.groene, peter.tabeling}@hpi.uni-potsdam.de`

Abstract

This paper addresses architectures of concurrent request processing servers, which are typically implemented using multitasking capabilities of an underlying operating system. Request processing servers should respond quickly to concurrent requests of an open number of clients without wasting server resources. This paper describes a small system of patterns for request processing servers, covering a relative wide range of architectures. Certain types of dependencies between the patterns are identified which are important for understanding and selecting the patterns. As the patterns deal with the conceptual architecture, they are mostly independent from programming languages and paradigms. The examples presented in this paper show applications of typical pattern combinations which can be found in productive servers. The pattern language is completed by a simple pattern selection guideline. The systematical compilation of server patterns, together with the guideline, can help both choosing and evaluating a server architecture.

1 Introduction

1.1 Application Domain

The patterns discussed in this paper focus on *request processing* servers which offer services to an *open number* of clients simultaneously.

Requests and Sessions.

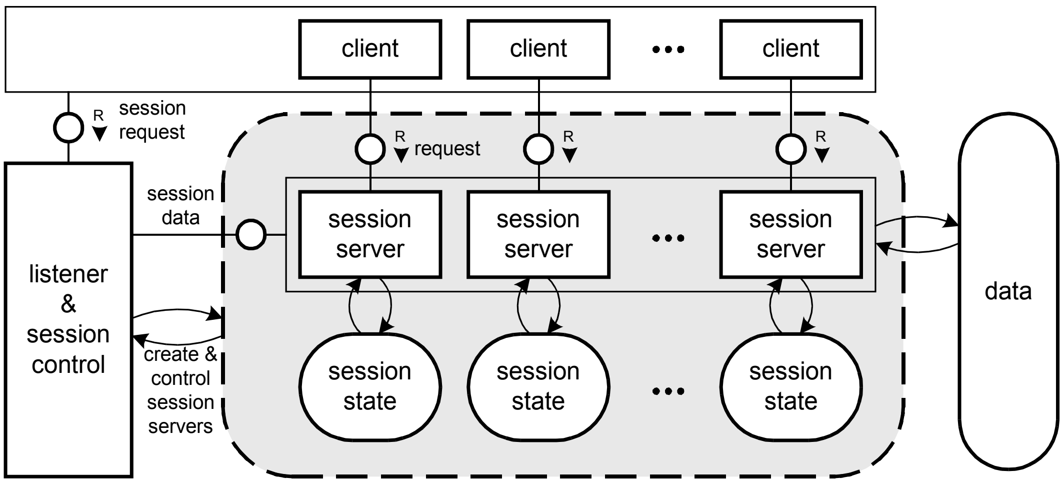

Figure 1 Clients and its corresponding servers (Block diagram legend: See figure 2)

Figure 1 presents an abstract conceptual model of the server type under consideration. For each client, it shows a dedicated *session server* inside the server. Each session server represents an abstract agent which exclusively handles requests of a corresponding client, holding the session–related state in a local storage. This abstract view leaves open if a session server is implemented by a task (process or thread) or not. A *session* starts with the first request of the client and represents the context of all further requests of the client until either the client or the server decides to finish it, which usually depends on the service and its protocol. Because each session server is only needed for the duration of the session it handles, session servers can be created and removed by the session controller on demand.

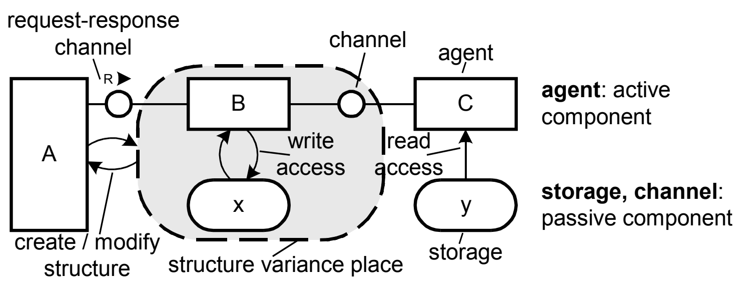

Figure 2 Legend of a block diagram

During a session, a client can send one or many requests to the server. For each request the server returns a response. In the following, we will exclusively focus on protocols where each client sends a sequence of requests and the server reacts with a response per request.

In this context, we have to distinguish between one-request-sessions and multiple-request-sessions. In case of *one-request-sessions*, the “session” is limited to the processing of only one single request, see figure 3. This is a typical feature of “stateless” protocols like HTTP. In contrast, a *multiple-request-session* like a FTP or a SMB session spans several requests, see figure 4. In this case, a session server repeatedly enters an idle state, keeping the session state for subsequent request processing until it is finally removed. If the client manages the session state, it sends the context to the server with every request, therefore the server can be simpler as it only has to manage single request sessions (The Keep Session data in the client pattern in [Sore02]), but this solution has many drawbacks. In this paper, we will focus on the server's point of view of multiple request sessions which results in the Keep Session data in the server pattern [Sore02].

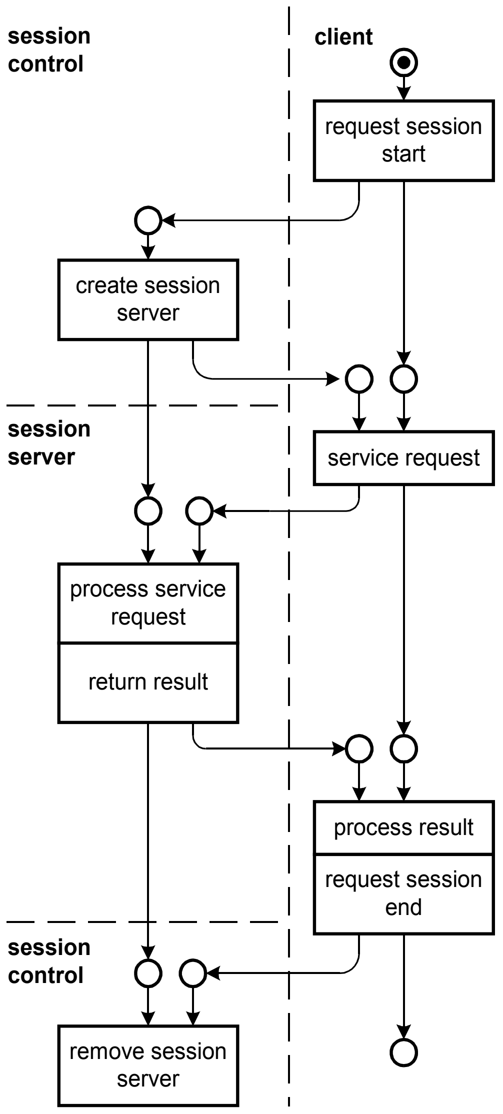

Figure 3 Single-request-session: Behavior of client, session server and session control (Petri Net)

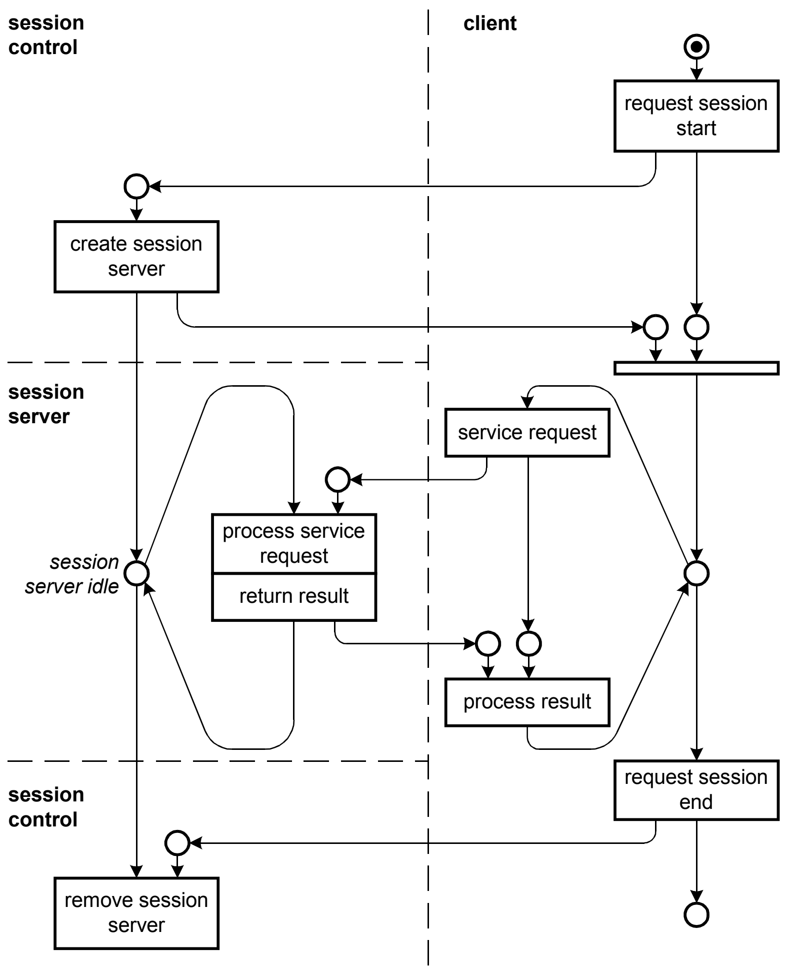

Figure 4 Multiple-request-session: Behavior of client, session server and session control (Petri Net)

Setting up Connections

If a dedicated connection between client and server (e.g. a TCP/IP connection) is used for communication during a session, the first request is a *connection request* sent by the client to the session controller. This request sets up the connection which can then be used to send *service requests* and return results. To set up the connection, the server has to accept the connection request. After establishing the connection, the TCP/IP service creates a new handle (socket) to access the new connection. The LISTENER / WORKER pattern shows this behavior in detail.

Multitasking

To serve multiple clients simultaneously, a server is usually implemented using processes or threads (tasks) of a given operating system. While the maximum number of concurrent clients is open in principle, it is actually constrained by the resource limitations of the server platform.

1.2 Forces

There are some forces which are common to all patterns described in this paper:

Response time. A network server should accept connection requests almost immediately and process requests as fast as possible. From the client's point of view, these time intervals matter:

1. Connect time (t_{conn}): The time from sending a connection request until the connection has been established.

2. First response time (t_{res1}): The time between sending the first request and receiving the response.
3. Next response times (t_{res2+}): The time between sending a subsequent request using an established connection and receiving a response.

The connect time t_{conn} usually should be short, especially for interactive systems. Minimizing the first response time t_{res1} is important for single-request sessions.

Limited Resources. Processes or threads, even when suspended, consume resources such as memory, task control blocks, database or network connections. Hence, it might be necessary to minimize the number of processes or threads.

Controlling the server. The administrator wants to shut down or restart the server without having to care for all processes or threads belonging to the server. Usually one of them is responsible for all other processes or threads of the server. This one receives the administrator's commands and may need some bookkeeping of the processes or threads belonging to the server.

1.3 Pattern Form

The pattern form used in this paper follows the Canonical Form. The graphical representations of architectural models are important parts of the solution's description. These diagrams follow the syntax and semantics of the Fundamental Modeling Concepts[1], while UML diagrams [UML] and code fragments are included as examples or hints for possible implementations. In order to reflect the cohesion within the pattern system, pattern dependencies (as discussed below) are explicitely identified.

1.4 Conceptual Focus of the Pattern Language

The patterns presented in this paper are not design patterns [GHJV94] in the narrow sense, i.e. they do not suggest a certain program structure such as a set of interfaces or classes. Instead, each pattern's solution is described as (part of) a *conceptual architecture*, i.e. as a dynamic system consisting of active and passive components without implying the actual implementation in terms of code elements written in some programming language. The resulting models mostly focus on the "conceptual view" or "execution view" according to [HNS99], but not the "module view" or "code view". While this approach leaves some (design) burden for a developer, it allows presenting the patterns in their wide applicability — they are not limited to a certain programming paradigm, but in practice can be found in a variety of implementations, including non–object oriented systems.

Because the patterns are related to a common application domain, they form a system with strong dependencies between the patterns: A pattern language. In this paper, three basic dependency types are relevant:

Independent patterns share a common application domain but address different problems. Both can be applied within the same project.

Alternative patterns present different solutions for the *same* problem. Only one of the two patterns can be applied, depending on which of the force(s) comes out to be the *dominant* one(s). In case of such patterns, the dominant force(s) is/are identified as such and the alternative patterns are put in contrast.

Consecutive patterns: In this case, pattern B (the consecutive pattern) can only be applied if pattern A has already been applied/chosen, i.e. the resulting context of A is the (given) context of B. Such patterns are described in order of applicability (A first, then B) with B explicitely referencing to A as a pre-requisite. This aids sorting out "secondary" patterns which become relevant at later stages or won't be applicable at all.

1 See [KTA+02]],[KeWe03],[Tabe02],[FMC]

2 A Pattern Language for Request Processing Servers

2.1 Overview

In the following, a system of seven patterns for request processing servers is presented. Table 1 and figure 5 give an overview. The Listener /Worker describes the separation of connection request and service request processing. Then there are two alternative task usage patterns, namely FORKING SERVER and WORKER POOL. For the latter, two alternative job transfer patterns are presented which are consecutive to the WORKER POOL pattern — JOB QUEUE and LEADER/FOLLOWER. As an additional, independent pattern, the SESSION CONTEXT MANAGER is discussed.

Pattern Name	*Problem*	*Solution*
LISTENER / WORKER	How do you use processes or threads to implement a server offering services for an open number of clients concurrently?	Provide different tasks for listening to and processing requests: One listener for connection requests and many workers for the service requests
FORKING SERVER	You need a simple implementation of the LISTENER / WORKER server. How can this server respond to an open number of concurrent requests in a simple way without using many resources?	Provide a master server listening to connection requests. After accepting and establishing a connection, the master server creates (forks) a child server which reads the request from this connection, processes the request, sends the response and terminates. In the meantime, the master server listens for connection requests again.
WORKER POOL	How can you implement a LISTENER / WORKER server providing a short response time?	Provide a pool of idle worker processes or threads ready to process a request. Use a mutex or another means to resume the next idle worker when the listener receives a request. A worker processes a request and becomes idle again, which means that his task is suspended.
WORKER POOL MANAGER	How do you manage and monitor the workers in a WORKER POOL?	Provide a WORKER POOL MANAGER who creates and terminates the workers and controls their status using shared worker pool management data. To save resources, the Worker Pool Manager can adapt the number of workers to the server load.
JOB QUEUE	How do you hand over connection data from listener to worker in a WORKER POOL server and keep the listener latency time low?	Provide a JOB QUEUE between the listener and the idle worker. The listener pushes a new job, the connection data, into the queue. The idle worker next in line fetches a job from the queue, reads the service request using the connection, processes it and sends a response back to the client.
LEADER / FOLLOWERS	How do you hand over connection data from listener to worker in a WORKER POOL server using operating system processes? How do you keep the handover time low?	Let the listener process the request himself by changing his role to "worker" after receiving a connection request. The idle worker next in line becomes the new listener while the old listener reads the request, processes it and sends a response back to the client.
SESSION CONTEXT MANAGER	How does a worker get the session context data for his current request if there are multiple-request-sessions and he just processed the request of another client?	Introduce a *session context manager*. Identify the session by the connection or by a session ID sent with the request. The session identifier is used by the session context manager to store and retrieve the session context as needed.

Table 1: Pattern Thumbnails

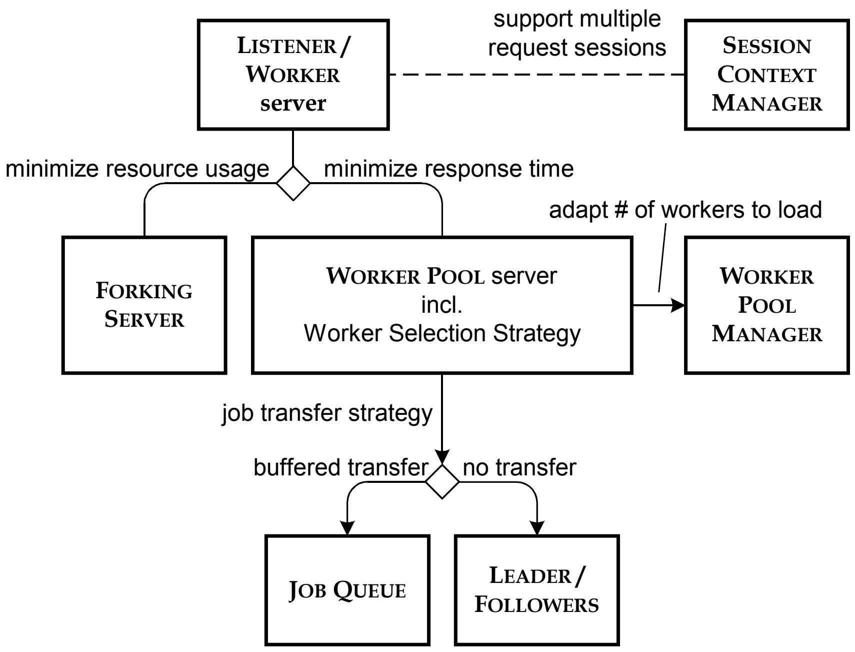

Figure 5 Overview of the Patterns in this document

2.2 Patterns for Request Processing Servers in Literature

For this domain, several patterns of this system have already been published in different forms. A good source is [POSA2] — in fact, most patterns described in this paper can be found in this book in some form. Some, like the Leader/Followers pattern, can be found directly, others only appear as variant (Half-Sync/Half-Reactive) or are mentioned as part of one pattern although they could be considered as patterns of their own (like the Thread Pool in Leader / Followers).

Apart from books, some pattern papers published for PLoP workshops cover aspects of request processing servers: The Request Handler pattern [VKZ02] describes what client and server have to do to post and reply to requests in general. The Pooling [KiJa02a], Lazy Acquisition [KiJa02b] and Eager Acquisition patterns describe aspects of the Worker Pool mechanisms. The Keep Session Data in Server and Session Scope Patterns [Sore02] are related to the Session Context Manager.

2.3 The Patterns

On the following pages, we will present the patterns as described above, each pattern starting on a new page. The guideline in section 2.4 gives hints when to choose which pattern.

Listener / Worker

Context

You want to offer services to an open number of clients using connection-oriented networking (for example TCP/IP). You use a multitasking operating system.

Problem

How do you use tasks (processes or threads) to implement a server offering services for an open number of clients concurrently?

Forces

- It is important to keep the time between connection request and establishing the connection small (connect time t_{conn}). No connection request should be refused as long as there is any computing capacity left on the server machine.
- For each server port there is only one network resource, a socket, available to all processes or threads. You have to make sure that only one task has access to a server port.

Solution

Provide different tasks for listening to connection requests and processing service requests:

- One *listener* listens to a server port and establishes a connection to the client after receiving a connection request.
- A *worker* uses the connection to the client to receive its service requests and process them. Many workers can work in parallel, each can process a request from a client.

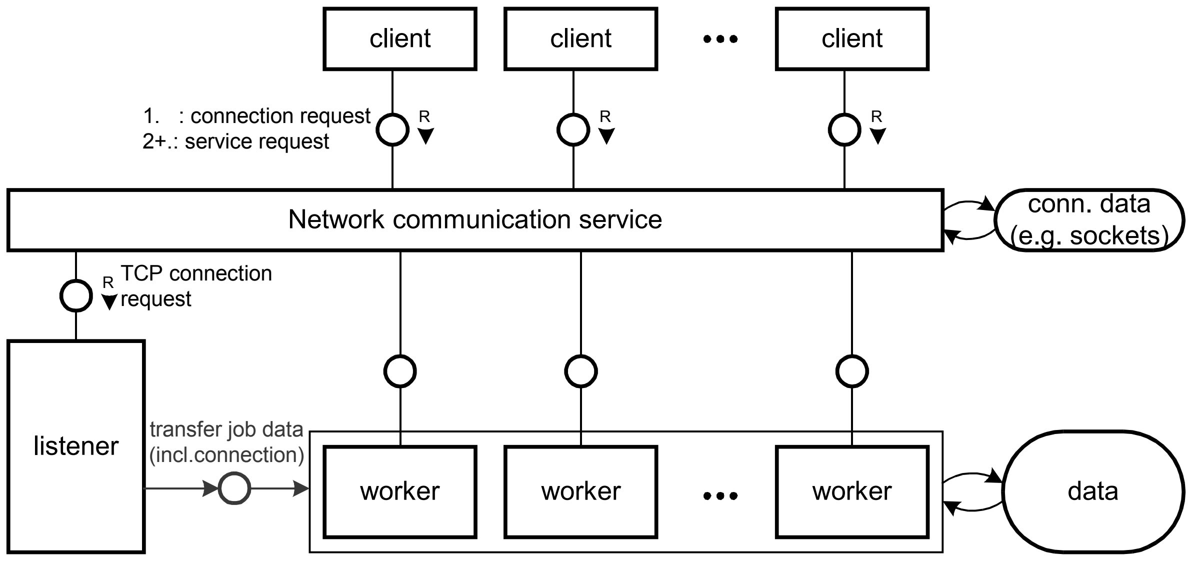

Figure 6 Listener / Worker pattern

Consequences

Benefits: As the listener's only task is to accept connection requests, the server is able to respond to connection requests quickly. Therefore it doesn't matter much that there is only one listener task per server port or even for all ports. A client sends its connection request and encounters that the connection will be established quickly and that he is connected to a worker exclusively listening to his request.

Liabilities: Although it is obvious that the listener has to exist with the server's start, you can still decide when to create the workers. You can either choose the FORKING SERVER pattern where the listener creates the worker on demand, that is for every connection request. Or choose the WORKER POOL pattern and create the workers in advance.

Transferring the job data (in this case, just the connection to the client) from listener to worker must also be implemented. For this, the patterns FORKING SERVER, JOB QUEUE and LEADER /FOLLOWERS offer three different solutions.

Response time. There are 4 time intervals to be considered for a LISTENER / WORKER Server:

1. Listener response time (t_1): The time the listener needs after receiving a connection request until he establishes the connection.
2. Listener latency (t_2): The time the listener needs after establishing a connection until he is again ready to listen.
3. Connection handover (t_3): The time between the listener establishes the connection and the worker is ready to receive the service request using this connection.
4. Worker response time (t_4): How long it takes for the worker from receiving a request until sending the response.

The values of t_1, t_2 and t_3 are only dependent from the multitasking strategy of the server, while t_4 is heavily dependent from the actual service request. All of them depend on the current server and network load, of course. Their effect on the response time intervals from the client's point of view (see section 1.2) is as follows: The listener needs t_1+t_2 for each connection request, so this sets his connection request rate and influences t_{conn}. The connection handover time t_3 is important for the first request on a new connection (t_{res1}); subsequent requests using this connection will be handled in t_4.

Known Uses

Most request processing servers on multitasking operating system platforms use the LISTENER / WORKER pattern. Some examples: The inetd, HTTP/FTP/SMB servers, the R/3 application server, database servers, etc.

Related Patterns

FORKING SERVER and WORKER POOL are two alternative consecutive patterns to address creation of the workers. SESSION CONTEXT MANAGER deals with the session data if workers should be able to alternately process requests of different sessions.

The ACCEPTOR—CONNECTOR pattern in [POSA2, p. 285] describes the separation of a server into acceptors and service handlers. The REACTOR pattern [POSA2, p. 179] is useful if one listener has to guard more than one port.

Forking Server

Context

You implement a LISTENER / WORKER server using tasks of the operating system.

Problem

You need a simple implementation of the LISTENER / WORKER server. How can this server respond to an open number of concurrent requests in a simple way without using many resources?

Forces

- Each operating system task (process or thread) consumes resources like memory and CPU cycles. Each unused and suspended task is a waste of resources.
- Transferring connection data (the newly established connection to the client) from listener to worker can be a problem if they are implemented with operating system processes.

Solution

Provide a master server listening to connection requests. After accepting and establishing a connection, the master server creates (forks) a child server which reads the service request from this connection, processes the request, sends the response and terminates. In the meantime, the master server listens for connection requests again. The forking server provides a worker task for each client's connection.

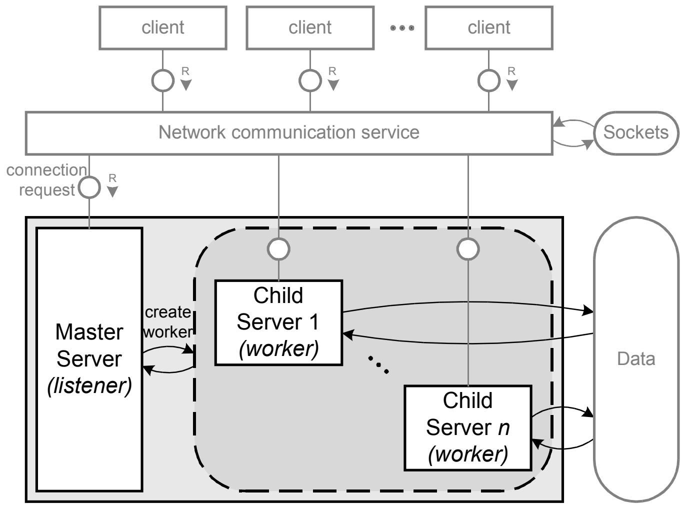

Figure 7 The forking server pattern

Figure 7 shows the runtime system structure of the FORKING SERVER. The structure variance area (inside the dashed line) indicates that the number of Child Servers varies and that the Master Server creates new Child Servers.

The master server task is the listener who receives connection requests from clients. He accepts a connection request, establishes a connection and then executes a "fork" system call which creates another task, a child server that also has access to the connection socket. While the listener returns to wait for the next connection request, the new child server uses the connection to receive service requests from the client. Figure 8 shows this behavior.

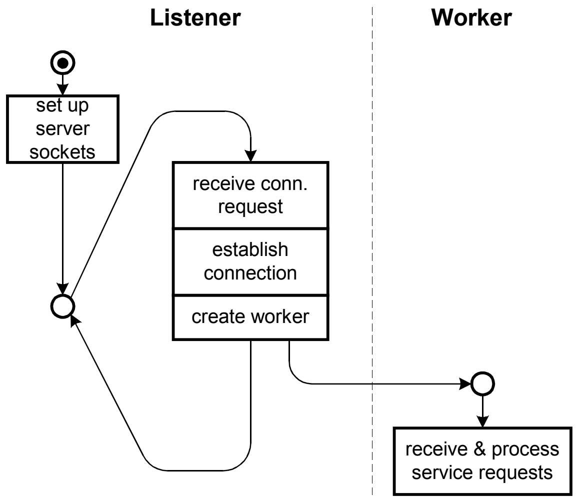

Figure 8 Behavior of the forking server

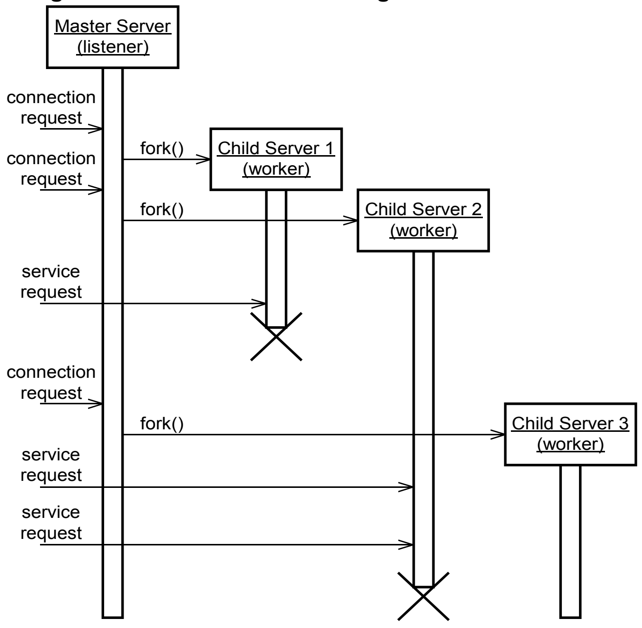

Figure 9 Example sequence of the forking server

Consequences

Benefits: The usage of server resources corresponds to the number of connected clients. A FORKING SERVER's implementation is simple:

- Connection handover: As `fork()` copies all process's data from parent to child process, this also includes the new connection to the client. If tasks are implemented with threads, it's even simpler because all threads of a process share connection handles.
- An idle FORKING SERVER needs very little system resources as it creates tasks on demand only.
- The Master Server only needs to know which workers are not terminated yet – this aids in limiting the total number of sessions (and therfore active workers) and makes shutting down the server quite simple.
- Handling multiple request sessions is easy because a worker can keep the session context and handle all service requests of a client exclusively until the session terminates.

Liabilities: A severe drawback of this kind of server is its response time. Creating a new task takes some time depending on the current server load. This will increase both the listener latency time t_2 and the job handover time t_3, which results in a bad connection response time and (first) request response time. If you need a more stable response time, use the WORKER POOL.

This also applies if you want to limit resource allocation and provide less workers than client connections. In this case you need a scheduler for the workers and a context management, for example the SESSION CONTEXT MANAGER.

Known uses

Internet Daemon. The Internet Daemon (inetd) is *the* prototype for the forking server which starts handlers for many different protocol types like FTP, Telnet, CVS — see section 3.1.

Samba smbd: Using the smbd, a Unix server provides Windows networking (file and printer shares) to clients. The Samba server forks a server process for every client.

Common Gateway Interface (CGI): An HTTP server which receives a request for a CGI program forks a new process executing the CGI program for every request.

Related Patterns

The WORKER POOL pattern offers an *alternative* solution, if a short response time is a critical issue. The SESSION CONTEXT MANAGER is an *independent* pattern which can be combined with FORKING SERVER, if a session spans multiple requests and it is not desirable to keep the according worker task for the complete session (for example, because the session lasts several hours or days).

The THREAD–PER–REQUEST pattern in [PeSo97] is very similar to the FORKING SERVER. The THREAD–PER–SESSION pattern in [PeSo97] describes the solution where session data is kept in the worker task instead of using a SESSION CONTEXT MANAGER.

Example Code

```
while (TRUE) {
  /* Wait for a connection request */
  newSocket = accept(ServerSocket, ...);
  pid = fork();
  if ( pid == 0 )
  {
    /* Child Process: worker */
    process_request(NewSocket);
    exit(0);
  }
  [...]
}
```

Worker Pool

Context

You implement a Listener / Worker server using tasks of the operating system.

Problem

How can you implement a Listener / Worker server providing a short response time?

Forces

- To minimize the listener latency time t_2, you have to make sure that the listener is quickly ready to receive new connection requests after having established a connection. Any actions that could block the listener in this phase increase the listener latency time.
- Creating a new worker task after establishing the connection increases the connection handover time t_3.

Solution

Provide a pool of idle worker tasks ready to process a request. Use a mutex or another means to resume the next idle worker when the listener receives a request. A worker processes a request and becomes idle again, which means that his task is suspended.

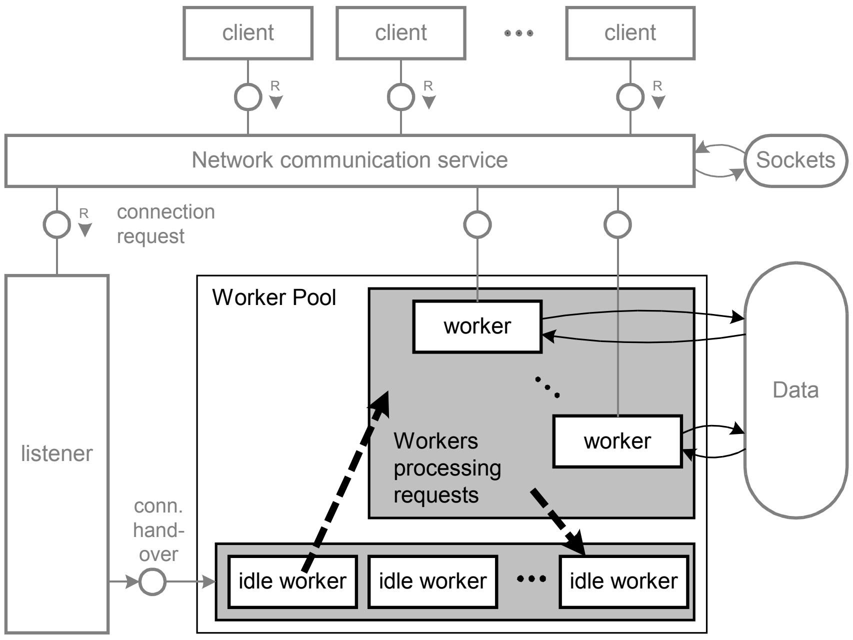

Figure 10 The Worker Pool pattern

Additionally, the strategy in choosing the next worker can be customized to gain better performance. The straightforward way is using a mutex. This usually results in a FIFO order or a random order, depending on the operating system resource in use. Another strategy could implement a LIFO order to avoid paging of task contexts.

The Worker Pool pattern is a good solution if the server's *response time* should be minimized and if it can be afforded to keep processes or threads "alive" in a pool between requests. In contrast to the Forking Server, this avoids creating a process or thread for every session or request, which increases the response time.

Consequences

Benefits:

- As the worker tasks are created in advance, the time to create a task doesn't affect the response time anymore.
- You can limit the usage of server resources in case of a high server load. It is even possible to provide less workers than clients.

Liabilities:

- You need a way to hand over the connection data from the listener to an existing worker. This strategy will affect the listener latency time t_2 and the connection handover time t_3. The two alternatives to do so are the JOB QUEUE and the LEADER / FOLLOWERS patterns.
- A static number of workers in the pool might be a problem for a varying server load. To adapt the number of workers to the current server load, use a WORKER POOL MANAGER.

Known Uses

Apache Web Server. All variants of the Apache HTTP server use a WORKER POOL. Most of them use a WORKER POOL MANAGER to adapt the number of workers to the server load. See section 3.2 for further details.

SAP R/3. The application server architecture of SAP R/3 contains several so-called "work processes" which are created at a server's start-up and stay alive afterwards to process requests. Usually there are less work processes in the pool than clients. As R/3 sessions usually span multiple requests, the work processes use a SESSION CONTEXT MANAGER. See section 3.3 for a more detailed description.

Related Patterns

The FORKING SERVER pattern is an *alternative* pattern which minimizes resource consumption but increases response time. WORKER POOL MANAGER is a consecutive pattern which provides a manager to control the workers. The SESSION CONTEXT MANAGER is an *independent* pattern which can be combined with WORKER POOL if a session spans multiple requests. JOB QUEUE and LEADER / FOLLOWERS are consecutive patterns which deal with the transfer of job-related data from listener to worker.

A detailed description of a thread pool can be found in [SV96] and in the LEADER / FOLLOWERS pattern in [POSA2, p. 450ff]. It is also mentioned in a variant of the ACTIVE OBJECT pattern [POSA2, p. 393] .

The POOLING Pattern [KiJa02a] describes more general how to manage resources in a pool. The creation of idle worker tasks at start-up is an example of EAGER ACQUISITION [KiJa02b].

Worker Pool Manager

Context

You have applied the WORKER POOL pattern.

Problem

How do you manage and monitor the workers in a WORKER POOL?

Forces

- At the server start, a certain number of worker tasks (processes or threads) have to be created before the first request is received.
- To shut down the server, only one task should receive the shutdown signal which will then tell the others to shutdown too.
- If a worker dies, he must be replaced by a new one.
- Workers consume resources, so there should be a strategy to adapt resource usage to the server load without reducing server response time.

Solution

Provide a WORKER POOL MANAGER who creates and terminates the workers and controls their status using shared worker pool management data.

To save resources, the Worker Pool Manager can adapt the number of workers to the server load: If the number of idle workers is too low or no idle worker available, he creates new workers. If the number of idle workers is too high, he terminates some idle workers.

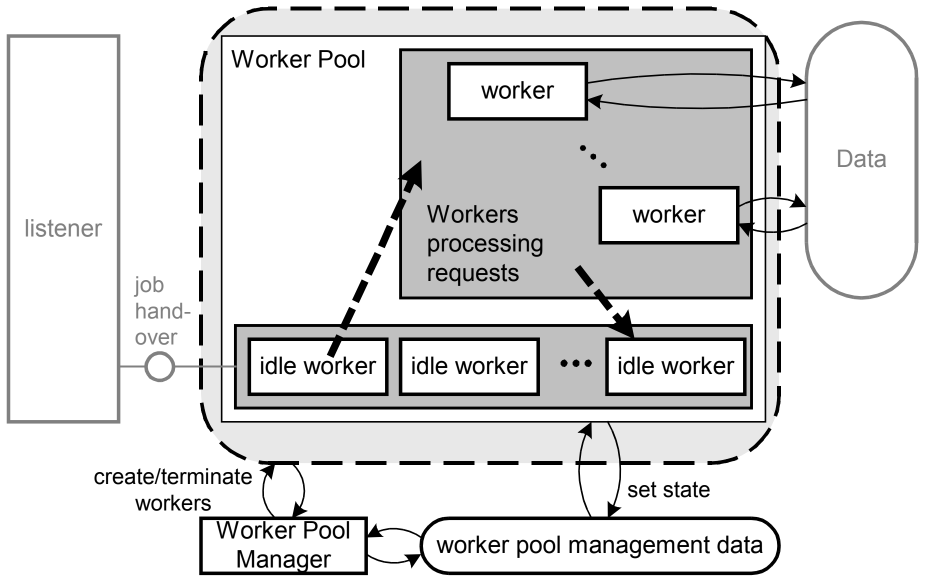

Figure 11 Worker Pool with manager

Figure 11 shows a WORKER POOL and its manager: The worker pool manager creates all tasks in the pool. For every task in the pool, it creates an entry in the worker pool management data. This storage is shared by the workers in the pool. Whenever an idle worker is activated or a worker becomes idle, it changes its state entry in the worker pool management data. The worker pool manager can count the number of idle and busy tasks and create new tasks or terminate idle tasks, depending on the server load. Additionally it can observe the "sanity" of the workers. If one doesn't show any life signs anymore or terminates because of a crash, the manager can replace it with a new one.

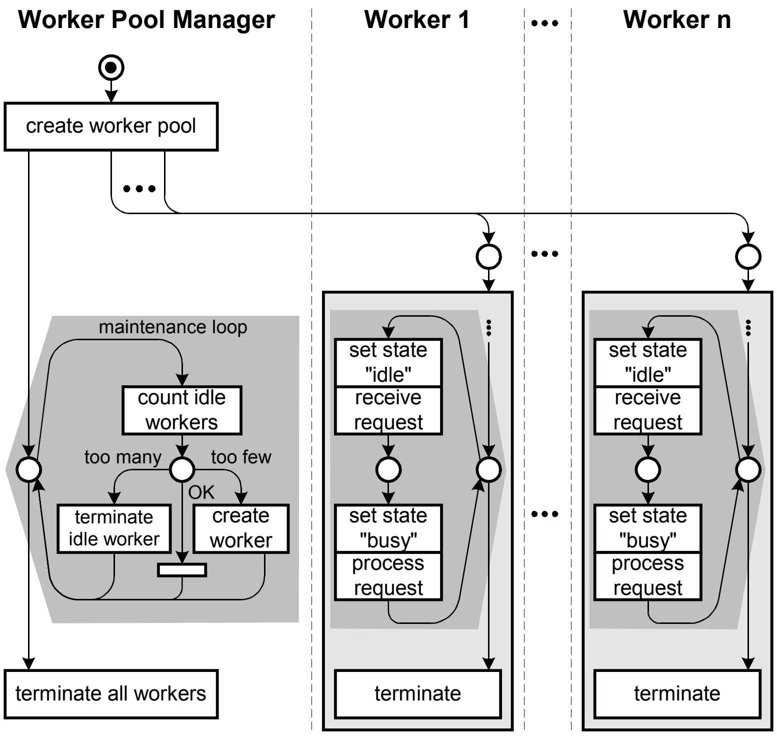

Figure 12 Interaction between Worker Pool Manager and Workers

Figure 12 shows the behavior of the worker pool manager and the workers: After creating the workers, the worker pool manager enters the maintenance loop where he counts the number of idle workers and terminates or creates workers, depending on the limits. The workers set their current state information in the worker pool management data.

Don't implement the WORKER POOL MANAGER in the same task as the listener, or you'll get the same problems as with the FORKING SERVER: Creating a new process may take some time which may increase the listener latency time t2 dramatically.

Figure 13 shows an example sequence where the worker pool manager replaces a worker which terminated unexpectedly.

Consequences

Benefits: The Worker Pool Manager takes care of the workers and makes it easier to shutdown a server in a well—defined way. By constantly monitoring the status of the workers, he helps to increase the stability of the server. If he controls the number of workers depending on the current server load, he helps to react to sudden changes in the server load and still keeps resource usage low.

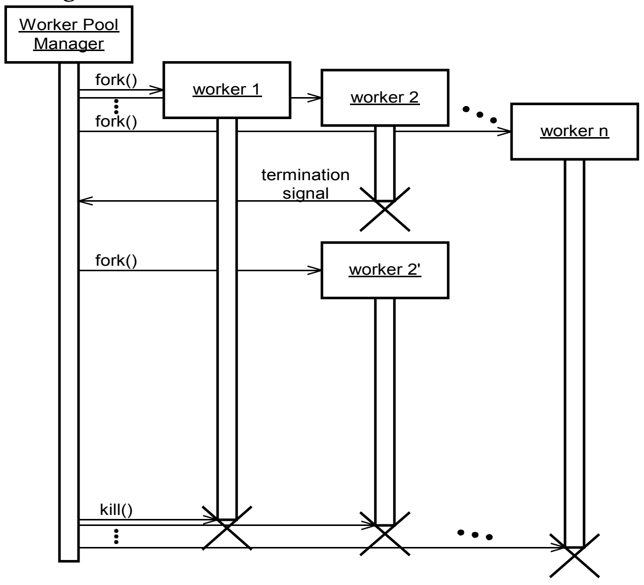

Figure 13 Example sequence: Worker Pool Manager creates, replaces and terminates workers

Liabilities: A worker now has to update its state entry in the worker pool management data storage whenever he enters another phase. This storage should be a shared memory. It is important to avoid blocking the workers while they update their state.

The Worker Pool Manager is an additional task which consumes resources as it has to run regularly.

Known Uses

Apache Web Server. All Apache MPMs have a dedicated process for worker pool management. Only the Windows version has a static worker pool while all other variants let the worker pool manager adapt the number of workers in the pool to the current server load.

SAP R/3 Dispatcher. The Dispatcher in an R/3 system manages the worker pool: He starts, shuts down and monitors the work processes and can even change their role in the system, depending on the current server load and the kind of pending requests.

Example Code

This example code has been taken from the Apache HTTP server (see section 3.2) and adapted to stress the main aspects of the pattern.

The `child_main()` code is not shown here as it depends on the job transfer strategy (Job Queue or Leader / Followers). Each worker updates his state in the scoreboard as shown in figure 12.

```
make_child(slot) {
[...]
  pid = fork();
  if ( pid == 0 )
  {
    /* Child Process: worker */
    scoreboard[slot].status = STARTING;
    child_main(slot);
    exit(0);
  }
  [...]
  scoreboard[i].pid = pid;
}

manage_worker_pool() {
  [...]
  scoreboard = create_scoreboard();
  /* create workers */
  for (i = 0 ; i < to_start ; ++i ) {
    make_child(i);
  }
  for (i = to_start ; i < limit ; ++i ) {
    scoreboard[i].status = DEAD;
  }

  /* Control Loop */
  while(!shutdown) {
    /* wait for termination signal or let some time pass */
    pid = wait_or_timeout();
    if ( pid != 0 ) {
      /* replace dead worker */
      slot = find_child_by_pid(pid);
      make_child(slot);
    }
    else {
      /* check number of workers */
      idle_count = 0;
      for (i = 0 ; i < limit ; ++i ) {
        if (scoreboard[i].status == IDLE)
        {
          idle_count++;
          to_kill = i;
        }
        if (scoreboard[i].status == DEAD)
          free_slot = i;
      }
      if (idle_count < min_idle) {
        make_child(free_slot);
      }
      if (idle_count > max_idle) {
        kill(scoreboard[to_kill].pid);
       scoreboard[to_kill].status = DEAD;
      }
    }
  } /* while(!shutdown) */

  /* Terminate server */
  for (i = 0 ; i < limit ; ++i ) {
    if (scoreboard[i].status != DEAD)
      kill(scoreboard[i].pid);
  }
}
```

Job Queue

Context

You have applied the WORKER POOL pattern to implement a LISTENER / WORKER server.

Problem

How do you hand over connection data from listener to worker in a WORKER POOL server and keep the listener latency time low?

Forces

- To decrease the listener latency time t_2, the listener should not have to wait for a worker to fetch a new job. This happens when the listener has just established a connection to the client and needs to hand over the connection data to a worker.

Solution

Provide a JOB QUEUE between the listener and the idle worker. The listener pushes a new job, the connection data, into the queue. The idle worker next in line fetches a job from the queue, reads the service request using the connection, processes it and sends a response back to the client.

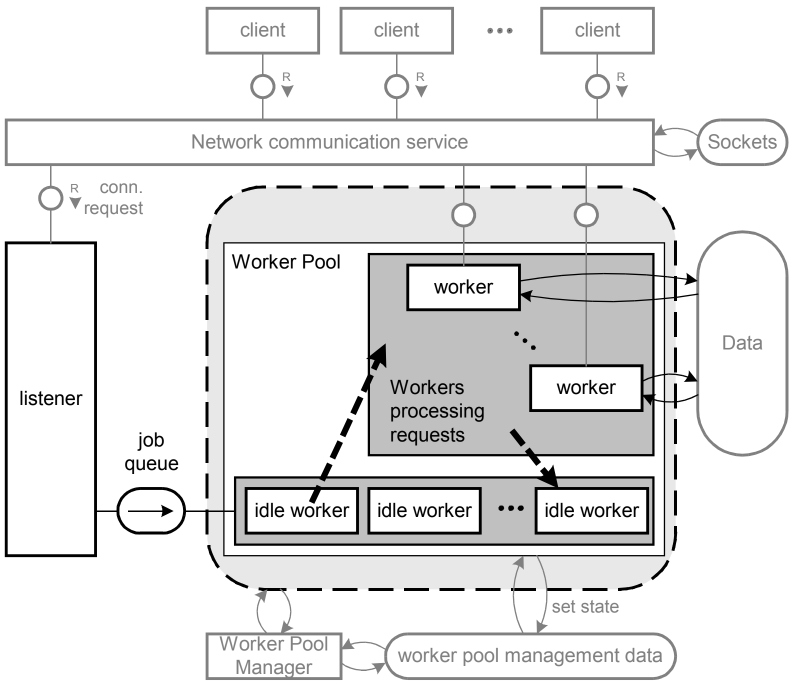

Figure 14 The job queue pattern

One or many listeners have access to the network sockets, either one listener per socket or one for all, as shown in figure 14. Instead of creating a worker task (like the Forking Server), he puts the connection data into the job queue and waits for the next connection request, see figure 15. All idle workers wait for a job in the queue. The first one to fetch a job waits for and processes service requests on that connection. After that, he becomes idle again and waits for his next job in the queue.

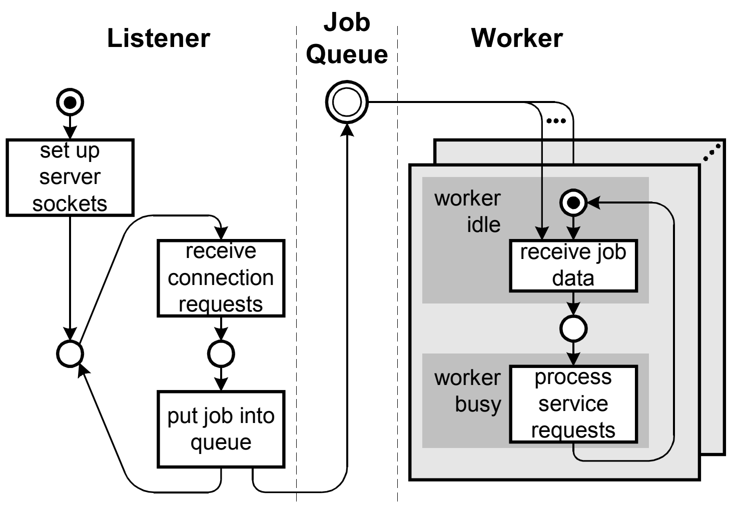

Figure 15 Behavior of Listener and Worker using a job queue

Consequences

Benefits: The listener just has to push a job into a queue and then return to listen again. The listener latency time t_2 is therefore low.

Liabilities: The job handover time t_3 is not optimal as the operating system has to switch tasks between listener, worker and queue mutex. When using operating system processes, the JOB QUEUE is not applicable, as there is no way to transfer a socket file descriptor (corresponding to the connection to the client) between two processes. In both cases, use the LEADER / FOLLOWERS pattern.

Known Uses

Apache Web Server. The Windows version (since Apache 1.3) and the WinNT MPM implement the JOB QUEUE with a fix number of worker threads. The worker MPM uses a JOB QUEUE on thread level (inside each process) while the processes concurrently apply for the server sockets using a mutex (section 3.2).

SAP R/3. On each application server of an R/3 system, requests are placed into a queue. These requests are removed from the queue by the "work proceses" for job processing, see section 3.3.

Related Patterns

JOB QUEUE is applicable as *consecutive* pattern to the WORKER POOL pattern. LEADER / FOLLOWERS is an *alternative* pattern which does not introduce a queue and avoids the transfer of job-related data between tasks.

The HALF-SYNC / HALF REACTIVE pattern in [POSA2, p.440] describes the mechanism to de-couple the listener (asynchronous service, reacting to network events) from the workers (synchronous services) using a message queue combined with the thread pool variant of the ACTIVE OBJECT pattern [POSA2, p. 393]. A description of the THREAD POOL with JOB QUEUE can be found in [SV96], including an evaluation of some implementation variants (C, C++ and CORBA).

Example Code

This example only shows the code executed in the listener and worker threads. The creation of the threads and the queue is not shown here.

The *job queue* transports the file descriptor of the connection socket to the workers and servers as a means to select the next worker.

Listener Thread:

```
while(TRUE) {
  [...]
  /* wait for connection request */
  NewSocket = accept(ServerSocket, ...);
  /* put job into job queue */
  queue_push(job_queue, NewSocket);
}
```

Worker Thread:

```
while (TRUE) {
  /* idle worker: wait for job */
  scoreboard[myslot].status = IDLE;
  [...]
  ConnSocket = queue_pop(job_queue);

  /* worker: process request */
  scoreboard[myslot].status = BUSY;
  process_request(ConnSocket);
}
```

Leader / Followers

Context

You have applied the Worker Pool pattern to implement a Listener / Worker server.

Problem

How do you hand over connection data from listener to worker in a Worker Pool server using operating system processes? How do you keep the handover time low?

Forces

- To access a new connection to the client, a task has to use a file descriptor which is bound to the process. It is not possible to transfer a file descriptor between processes.
- Switching tasks (processes or threads) between listener and worker increases the connection handover time t_3.

Solution

Let the listener process the service request himself by changing his role to ″worker“ after receiving a connection request. The idle worker next in line becomes the new listener while the old listener reads the service request, processes it and sends a response back to the client.

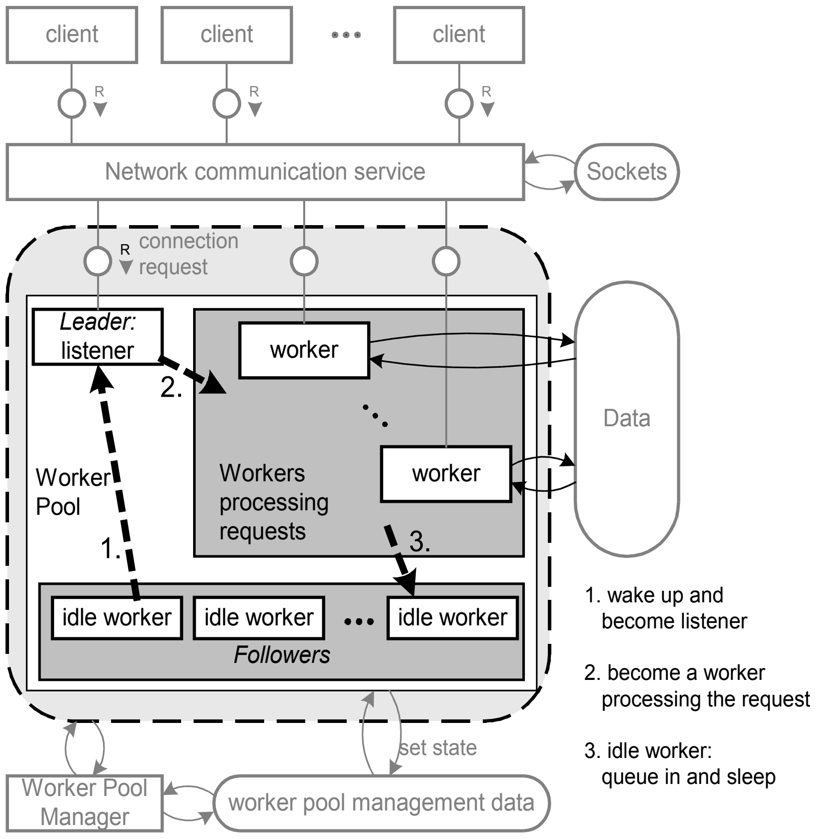
client
client
client
R
Network communication service
Sockets
connection request
Leader: listener
worker
worker
Worker Pool
Workers processing requests
Data
idle worker
idle worker
idle worker
Followers
1.
2.
3.
1. wake up and become listener
2. become a worker processing the request
3. idle worker: queue in and sleep
set state
Worker Pool Manager
worker pool management data

Figure 16 The Leader / Followers pattern

All tasks in the worker pool transmute from listener to worker to idle worker eliminating the need for a job queue: Using a mutex, the idle workers (the followers) try to become the listener (the leader). After receiving a connection request, the listener establishes the connection, releases the mutex and becomes a worker processing the service request he then receives. Afterwards, he becomes an idle worker and tries to get the mutex to become the leader again. Hence, there is no need to transport information about the connection as the listener transmutes into a worker task keeping this information.

Figure 16 shows the structure: The processes or threads in the WORKER POOL have 3 different states: worker, idle worker and listener. The listener is the one to react to connection requests, while workers and idle workers process service requests or wait for new jobs, respectively.

The corresponding dynamics are shown in figure 17. Initially, all tasks of the pool are idle workers. Listener selection is done by a task acquiring the mutex which is released as soon as the listener changes his role to become a worker.

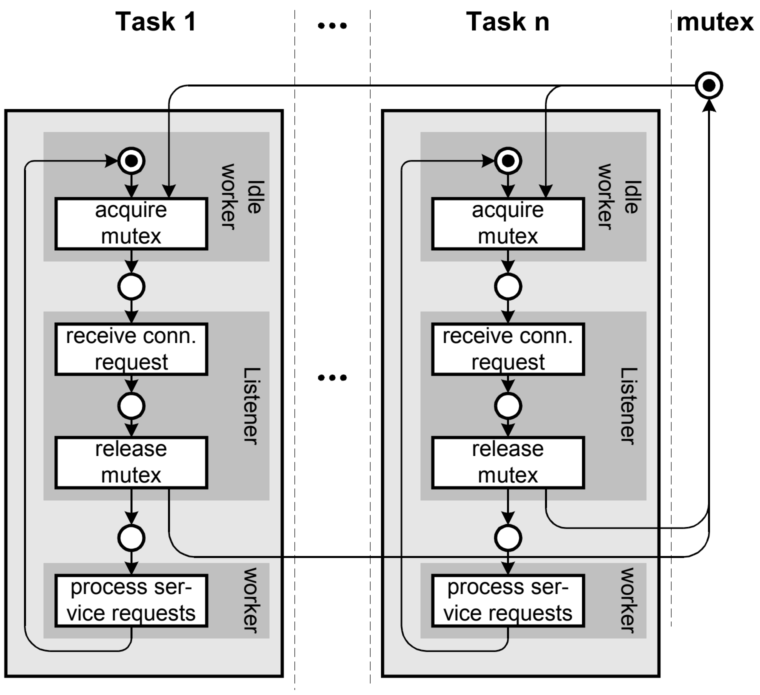

Figure 17 Behavior of the tasks in the Leader / Followers pattern

Consequences

Benefits: The listener changes his role by just executing the worker's code. This keeps the connection handover time t_3 very low as every information remains in this task. As file descriptors needed to get access to the connection to the clients don't leave the task, the LEADER /FOLLOWERS pattern enables the use of operating system processes to implement the WORKER POOL pattern.

Liabilities: The election of the next listener is handled via a mutex or a similar mechanism. This requires a task switch and leads to a non-optimal listener latency time t_2. The LEADER / FOLLOWERS pattern avoids transferring job-related data, but introduces the problem of dynamically changing a process' or thread's role; for example, the server sockets must be accessible to all workers to allow each of them to be the listener. Both listener and worker functionality must be implemented inside one task.

Known Uses

Apache Web Server. A very prominent user of LEADER / FOLLOWERS pattern is the preforking variant of the Apache HTTP server (see section 3.2).

Call center. In a customer support center, an employee has to respond to support requests of customers. The customer's call will be received by the next available employee.

Taxi stands. Taxis form a queue at an airport or a train station. The taxi at the top of the queue gets the next customers while the others have to wait.

Related Patterns

LEADER / FOLLOWERS is applicable as *consecutive* pattern to the WORKER POOL pattern. JOB QUEUE is an *alternative* pattern which uses a queue for job transfer between tasks with static roles.

The LEADER / FOLLOWERS pattern has originally been described in [POSA2], p. 447.

Example code:

```
while (1) {
  /* Become idle worker */
  scoreboard[myslot].status = IDLE;
  [...]
  acquire_mutex(accept_mutex);
  /* Got mutex! Now I'm Listener! */
  scoreboard[myslot].status = LISTENING;
  newSocket = accept(ServerSocket, ...);
  [...]
  /* Become worker ... */
  release_mutex(accept_mutex);
  /* ... and process request */
  scoreboard[myslot].status = BUSY;
  process_request(NewSocket);
}
```

Session Context Manager

Context

You implement a LISTENER / WORKER server for multiple-request sessions.

Problem

If a worker processes service requests from different clients and a session can contain multiple requests, how does a worker get the session context data for his current service request?

Forces

Keeping session context data within a worker can be a problem:

- If a worker task is assigned exclusively for one session, it is unnable to handle requests from other clients. This is usually a waste of resources and interferes with limiting the number of workers.
- You have to consider that the connection to the client may be interrupted during the session without terminating the session. The same applies to the worker task which can die unexpectedly. Therefore you might need to save and resume a session state.

Solution

Introduce a *session context manager*. Identify the session by the connection or by a session ID sent with the request. The session identifier is used by the session context manager to store and retrieve the session context as needed.

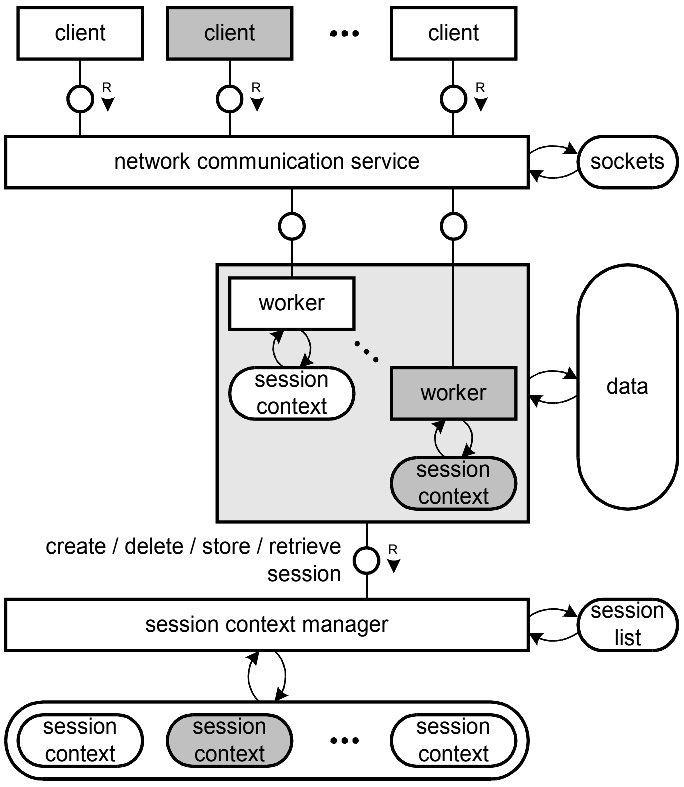

Figure 18 Session Context Manager (Central Manager Variant).

This is a specialized variant of the MEMENTO pattern [GHJV94] . Figure 18 shows a solution using a central session context manager: Each worker has a local storage for a session context. Before he processes a request, he asks the context manager to retrieve the session context corresponding to the client's session using the session ID extracted from the request. After processing the request, he asks the context manager to store the altered session context. In case of a new session, he has to create a new session context and assign a session ID to it. In case of the last request of a session, the worker has to delete the session context. The session context shaded grey in figure 18 belongs to the grey client. The grey worker currently processes a request of this client and works on a copy of its session context.

Figure 19 shows how to extend the behavior of the worker to support session context management. An example sequence is shown in figure 20.

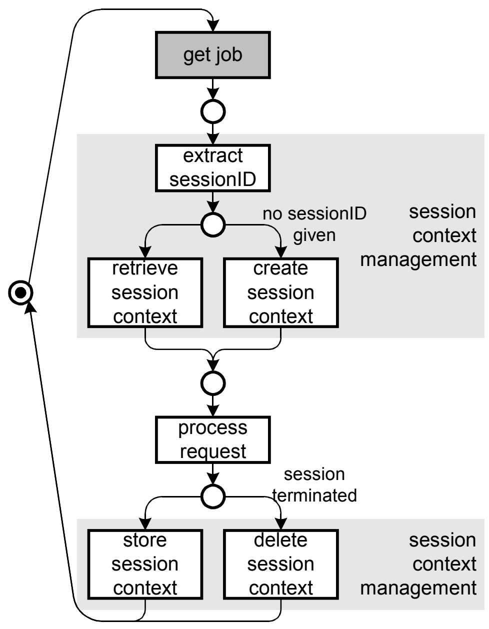

Figure 19 Session Context Management in the worker loop

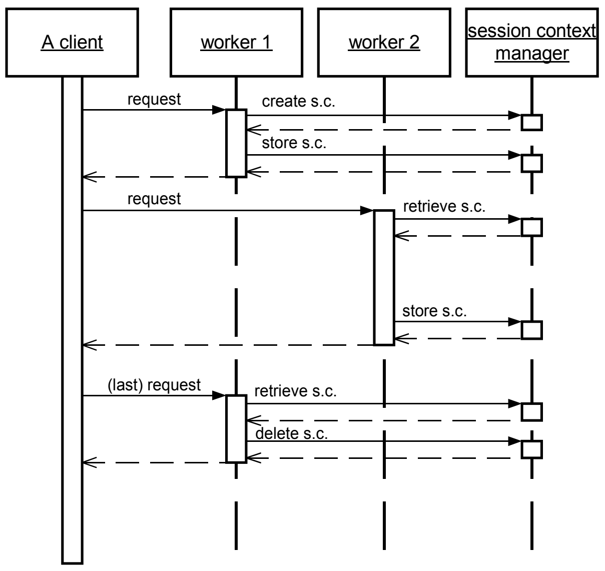

Figure 20 Example sequence for session context management

Variants. *Central Session Context Manager:* There is a single context manager for all tasks. If, for example, the session is bound to the connection, the listener not only reacts to connection requests but also to requests on open connections. The functionality of the session context manager can then be included in the listener.

Local Session Context Manager: Each worker manages the sessions and the session list. The functionality of the session context manager in figure 18 is then included in every worker task. The storage for the session contexts is shared between all workers.

Consequences

Benefits.

- Any worker can process a request in its session context. This enables an efficient usage of workers, especially in a WORKER POOL server.
- If the session ID is sent with every request, the connection can be interrupted during the session. This is useful for long sessions (from a few minutes to several days).
- Using a dedicated context manager helps separating the request-related and context-related aspects of request processing. For each request to be processed, session context management requires a sequence of (1) retrieving (creating) a session context, (2) job processing and (3) saving (deleting) the context.

Liabilities.

- A garbage collection strategy might be needed to remove session contexts of "orphan" sessions.
- Session context storage and retrieval increases the response time.

Known Uses

SAP R/3. Each application server of an SAP R/3 system contains workers, each of them having its own local session context manager (the so-called taskhandler, see section 3.3).

CORBA portable object adapter. CORBA-based servers can use objects (not tasks) as workers similar to a Worker Pool. In such configurations the so-called object adapters play the role of session context managers, see section 3.4.

CGI applications. An HTTP server starts a new CGI program for every request, like the FORKING SERVER. The CGI program extracts the session ID from the request (for example by reading a cookie) and then gets the session context from a file or database.

Related Patterns

The pattern is *consecutive* to FORKING SERVER or WORKER POOL. To realize access to session contexts of workers, the MEMENTO pattern [GHJV94] could be used. In case of local context management, TEMPLATE METHOD [GHJV94] could be applied to enforce the retrieve–process–save sequence for each request. The KEEP SESSION DATA IN SERVER and SESSION SCOPE Patterns [Sore02] describe session management in a more general context.

2.4 Guideline for Choosing an Architecture

In the following, a simple guideline for selecting patterns from the pattern system is presented. It helps deriving a server architecture by choosing a pattern combination appropriate for the intended server usage. The guideline presents the patterns according to their dependencies and fosters pattern selection by questions aiming at dominant forces.

1. Clarify the basic context for LISTENER / WORKER.
   - Is the server's type a *request processing server* or a different one, e.g. a streaming server? Should threads or processes provided by the operating system be used, including IPC mechanisms? If not, the pattern system might be not appropriate.
   - Does a session span *multiple requests*? Then consider 4.
2. Select the task usage pattern.

   Is saving *resources* more important than minimizing the *response time*? If yes, choose a FORKING SERVER. If not, apply the WORKER POOL pattern instead.
3. When using a WORKER POOL,
   - Choose a Job Transfer pattern. Is *transferring job data* between tasks easier than *changing their role*? If yes, introduce a JOB QUEUE. If not, apply the LEADER/FOLLOWER pattern.
   - Does the number of concurrent requests vary in a wide range? Then use a dynamic pool instead of a static pool. In this case the WORKER POOL MANAGER dynamically adapts the number of workers to the server load.
   - Choose a *strategy* to select the next worker task from the idle worker set: FIFO, LIFO, priority-based, indetermined.
4. If a session spans multiple requests:
   - Does the number of concurrent sessions or their duration allow to keep a worker task exclusively for the session? If not, introduce a SESSION CONTEXT MANAGER.
   - Can the listener retrieve the session ID of a client's request? Then choose central context management, else local.

The following table shows that the possible pattern combinations yield six basic architecture types:

		***without* SESSION CONTEXT MANAGER**	***with* SESSION CONTEXT MANAGER**
FORKING SERVER		***Type 1*** inetd, samba server, CGI	***Type 2*** CGI applications
WORKER POOL	**JOB QUEUE**	***Type 3*** Apache (Win32, Worker)	***Type 4*** SAP R/3
	LEADER/FOLLOWER	***Type 5*** Apache (Preforking)	***Type 6***

Table 2 Architectures covered by the pattern system

3 Example Applications

3.1 Internet Daemon

The Internet Daemon (inetd) is a typical representative of the Forking Server without Session Context Manager (*Type 1*).

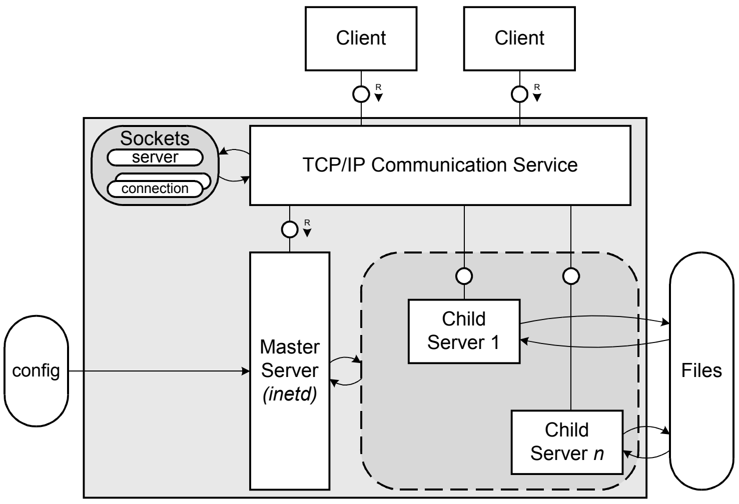

Figure 21 The inetd — a typical Forking Server

Figure 21 shows the structure of the inetd server: It waits for requests on a set of TCP ports defined in the configuration file `/etc/inetd.conf`. Whenever it receives a connection request, it starts the server program defined for this port in the configuration file which handles the request(s) of this connection. The inetd starts a server program by the `fork() – exec()` sequence which creates a new process and then loads the server program into the process. The file descriptor table is the only data which won't be deleted by `exec()`. A server program to be started by the inetd must therefore use the first file descriptor entry (#0) to access the connection socket.

3.2 Apache HTTP Server

The Apache HTTP server [Apache] is a typical request processing server which has been ported to many platforms. The early versions use the *preforking* server strategy as described below. Since version 1.3, Apache supports the Windows^{TM} platform using threads which forced another server strategy (Job Queue).

Apache 2 now offers a variety of server strategies called MPMs (Multi–Processing Modules) which adapts Apache to the multitasking capabilities of the platform and may offer different server strategies on one platform. The most interesting MPMs are:

- Preforking *(Type 5)*:
  Leader / Followers using processes with dynamic worker pool management. The promotion of the followers is done with a mutex (results in a FIFO order).
- WinNT *(Type 3)*:
  Job Queue using a static thread pool.
- Worker MPM (*Type 3* on thread level):
  Each process provides a Job Queue using a static thread pool. The process pool is dynamically adapted to the server load by a Worker Pool Manager (Master Server). The listener threads of the processes use a mutex to become listener. (see section 3.2)

All MPMs use a Worker Pool with processes or threads or even nest a thread pool in each process of a process pool. They separate the listener from the Worker Pool Manager. A so-called

Scoreboard is used to note the state of each worker task. Most Worker Pool managers adapt the number of worker tasks to the current server load.

A detailed description of the Apache MPMs and of their implementation can be found in the Apache Modeling Project [AMP].

The Preforking MPM of Apache

Since its early versions in 1995, Apache uses the so-called Preforking strategy — a LEADER / FOLLOWERS pattern. A master server starts (forks) a set of child server processes doing the actual server tasks: listen for connection requests, process service requests. The master server is responsible for adjusting the number of child servers to the server load by assuring that the number of idle child servers will remain within a given interval.

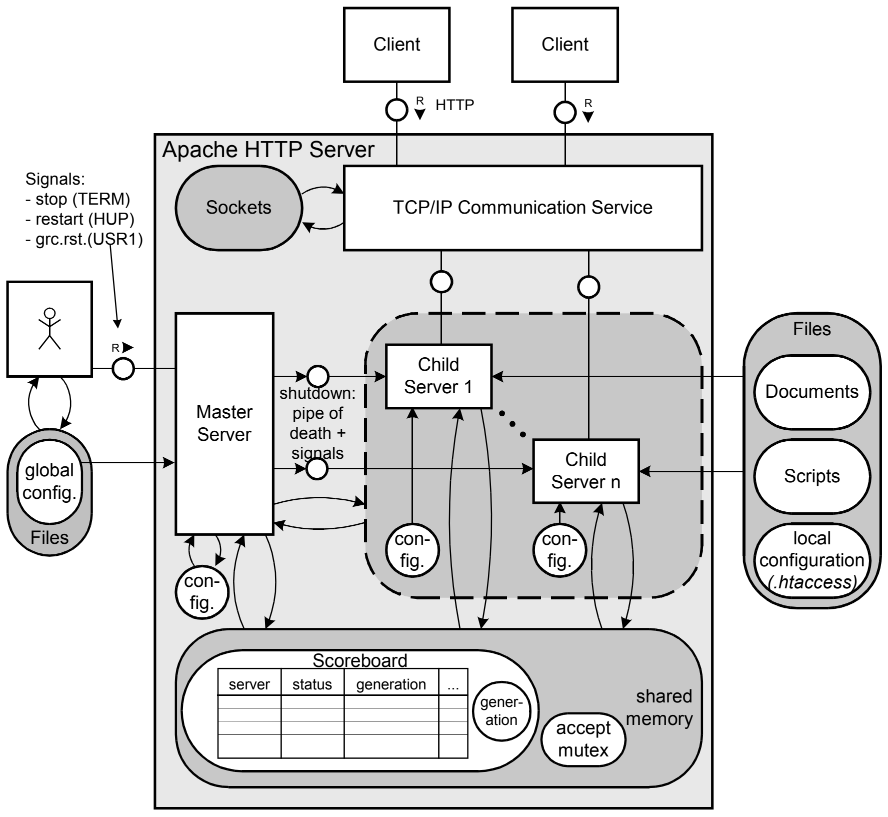

Figure 22 The Apache server using the Preforking MPM

Figure 22 shows the conceptual architecture of a preforking Apache server. At the top we see the clients, usually web browsers, sending HTTP requests via a TCP connection. HTTP is a stateless protocol, therefore there's no need to keep a session context. At the right-hand side we see the data to be served: Documents to be sent to the client or scripts to be executed which produce data to be sent to the client. The Scoreboard at the bottom keeps the Worker Pool management data as mentioned in the WORKER POOL pattern.

The master server is not involved in listening or request processing. Instead, he creates, controls and terminates the child server processes and reacts to the commands of the administrator (the agent at the left side). The master server also processes the configuration files and compiles the configuration data. Whenever he creates a child server process, the new process gets a copy of this configuration data. After the adminstrator has changed a configuration file, he has to advise the master server to re-read the configuration and replace the existing child servers with new ones including a copy of the new configuation data. It is possible to do this without interrupting busy child servers: The master server just terminates idle child servers and increments the generation number in the scoreboard. As every child server has an entry including its generation in the scoreboard, it checks after each request if its generation is equal to the current generation and terminates otherwise.

The child servers use a mutex to assign the next listener, according to the Leader / Followers pattern. In contrast to figure 16, the different roles of the workers in the pool are not shown.

The Worker MPM of Apache

Figure 23 shows the system structure of the Apache HTTP server using the worker MPM. The center shows multiple child server processes which all have an identical structure. They are the tasks of the Worker Pool on process level, like in the preforking MPM. Inside each Child Server Process we find an extended Job Queue structure: A listener thread waits for connection requests and supplies a job queue. (Idle) worker threads wait for new jobs. The idle worker queue signals to the listener if there is an idle worker ready to process the next job. If there is none, the listener doesn't apply to the accept mutex.

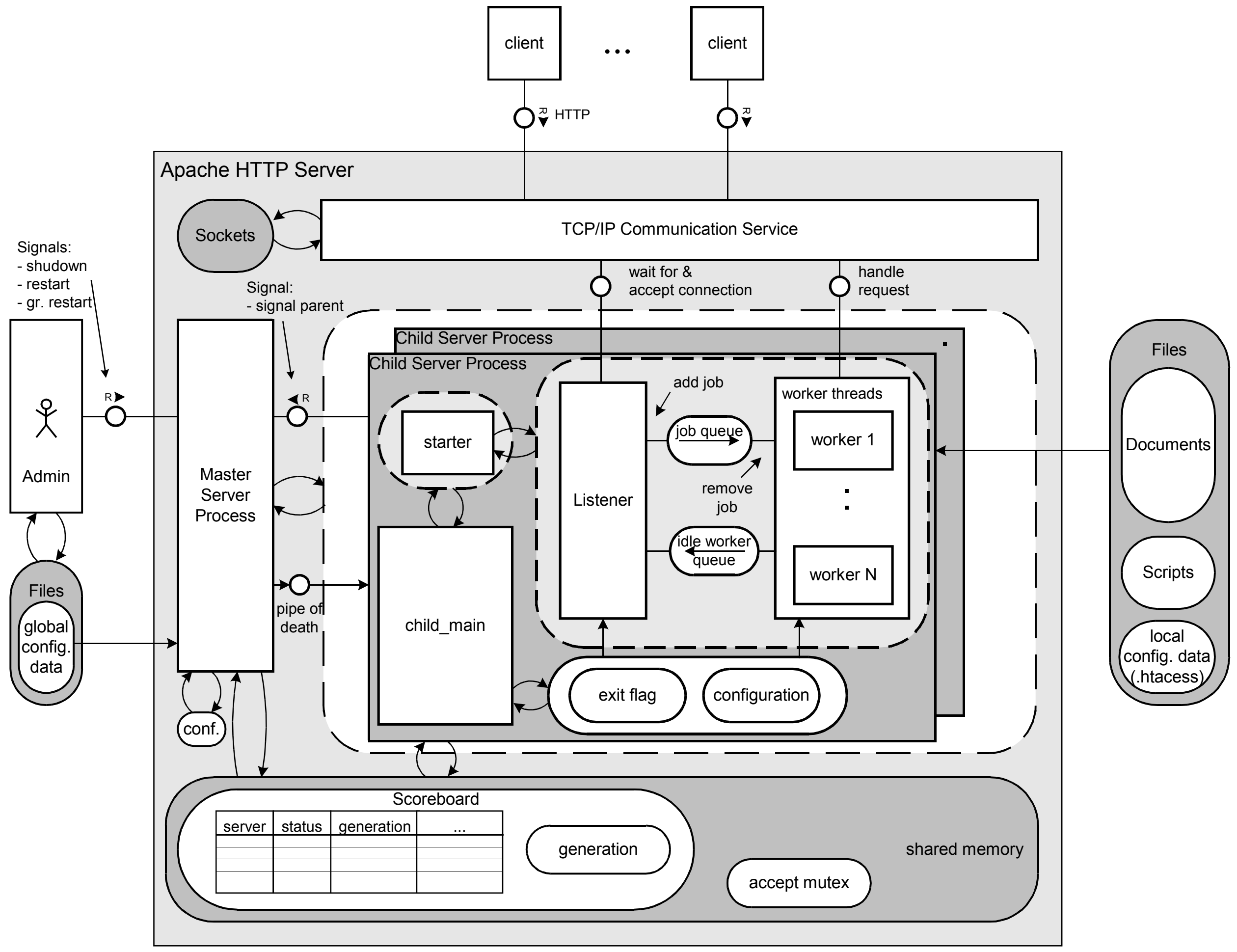

Figure 23 The Apache server using the Worker MPM

The child_main thread creates the listener and worker threads by creating the starter thread (which terminates after setting up listener, workers and the queues) and after that just waits for a termination token on the 'pipe of death'. This results in setting the “exit flag” which is being read by all threads.

Only one process' listener gets access to the server sockets. This is done by applying for the accept mutex. In contrast to the Leader / Followers pattern, the listener task doesn't change his role and processes the request. Instead, he checks if there is another idle worker waiting and applies for the accept mutex again.

3.3 SAP R/3

SAP's System R/3 [SAP94] is a scalable ERP system with an overall three tier architecture as shown in figure 24. The diagram also shows the inner architecture of an R/3 application server. (In practice, R/3 installations often include additional applications servers — these are omitted here for simplicity reasons.) The application server can be categorized as a *Type 4* server (cp. Table 2), because three of the patterns discussed above are actually applied.

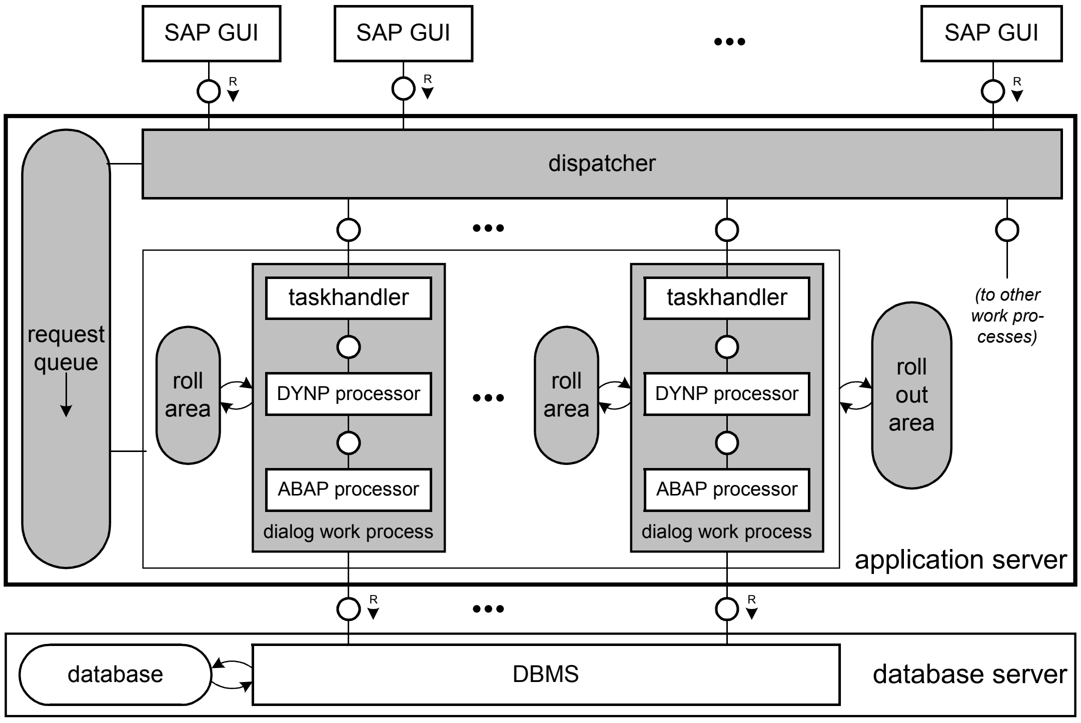

Figure 24 SAP R/3 Application Server Architecture

The server's basic architecture is designed after the WORKER POOL pattern. The so-called dispatcher (a process) plays the role of the listener and forwards requests (sent by the "SAP GUI" clients) to worker tasks called "dialog work processes". (There are further types of work processes, but these are of no interest here.)

Beside other purposes, the request queue is used for forwarding requests from the dispatcher to the work processes, i.e. it represents a JOB QUEUE. While the work processes read request data from the queue by themselves, initial worker selection is actually done by the dispatcher.

Because an R/3 session (called a *transaction* in SAP's terms) spans multiple requests, a local SESSION CONTEXT MANAGER called *taskhandler* is included in each *work process*. Before a request can be processed by the DYNP processor and the ABAP processor, the taskhandler retreives the appropriate context from the *roll out area* (or creates a new one) and stores it in the work process' *roll area*. Afterwards, the taskhandler saves the context to the roll out area (at session end, it would be deleted).

3.4 Related Applications at Object Level

All of the above patterns have been discussed under the assumption of processes or threads being the building blocks for server implementations. However, some of the patterns are even applicable if we look at a system at the level of *objects* instead of tasks. When realizing a server's request processing capabilities by a set of objects, these objects can be "pooled" similar to a WORKER POOL. Instead of creating objects for the duration of a session or a single request, "worker objects" are kept in a pool and activated on demand. This helps controlling resource consumption and avoids (potentially) expensive operations for creating or deleting objects.

For example, this idea has been put into practice with Sun's J2EE server architecture [J2EE]. Here, the "stateless session beans" are kept in a pool while the so-called "container" plays the listener role, activating beans on demand and forwarding requests to them.

Another example are the so-called "servants" from the CORBA 3.0 portable object adapter specification [CORBA]. These are server-side worker objects which can also be kept in a pool managed by the "object adapter" (if the right "server retention policy" has been chosen, see the POA section in [CORBA]). The object adapter (in cooperation with an optional "servant manager") does not only play the listener role — it also acts as SESSION CONTEXT MANAGER for the servants.

4 Conclusion and Further Research

Design patterns in the narrow sense are often discussed in a pure object-oriented context. Hence, they often present object-oriented code structures as solution, typically classes, interfaces or fragments of methods. In contrast, the patterns presented in this paper are conceptual patterns which deliberately leave open most of the coding problem. This initially seems to be a drawback, but it also widens the applicability of a pattern and increases the possibility to identify a pattern within a given system. In fact, most of the industrial applications (known uses) examined in this paper are not implemented with an object-oriented language (although some OO concepts can be found). Furthermore, central ideas and topics (e.g. scheduling, task and session management) behind the patterns have already been described in the literature about transaction processing systems [GrRe93].

Designing a good code structure is often a secondary problem with additional forces such as given languages, frameworks or legacy code. In order to remain "paradigm–neutral", conceptual architecture patterns should be presented using appropriate notations like FMC. Object–Oriented implementation of conceptual architecture models is an important research topic in this context [TaGr03].

The integrated description of pattern systems has been developed together with the pattern system presented here. Further research is necessary to prove that this is applicable to pattern systems in general. This approach, backed by corresponding guidelines, may support software architects in applying patterns.

5 Acknowledgements

We wish to thank Uwe Zdun who gave us many valuable hints as our shepherd, and the participants in the writer's workshop, especially Neil Harrison, who gave us lots of very useful advice and suggestions for improvement.

References

[Apache] Apache Group, *The Apache HTTP Server* , http://httpd.apache.org

[AMP] Bernhard Gröne et al., *The Apache Modeling Project*, http://apache.hpi.uni-potsdam.de

[CORBA] *The Common Object Request Broker Architecture*, version 3.0.2, The Object Management Group, 2002

[FMC] Siegfried Wendt et al: *The Fundamental Modeling Concepts* Home Page, http://fmc.hpi.uni-potsdam.de/

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[GrRe93] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993

[HNS99] Christine Hofmeister, Robert Nord and Dilip Soni, *Applied Software Architecture*, Addison Wesley, 1999

[J2EE] *Java 2 Platform: Enterprise Edition*, Sun Microsystems Inc., http://java.sun.com/j2ee

[KeWe03] Frank Keller, Siegfried Wendt, *FMC: An Approach Towards Architecture-Centric System Development*, IEEE Symposium and Workshop on Engineering of Computer Based Systems, Huntsville, 2003

[KiJa02a] Michael Kircher and Prashant Jain, *Pooling*, EuroPLoP 2002

[KiJa02b] Michael Kircher and Prashant Jain, *Eager Acquisition*, EuroPLoP 2002

[KTA+02] Frank Keller, Peter Tabeling, Rémy Apfelbacher, Bernhard Gröne, Andreas Knöpfel Rudolf Kugel and Oliver Schmidt, *Improving Knowledge Transfer at the Architectural Level: Concepts and Notations*, International Conference on Software Engineering Research and Practice (SERP), Las Vegas, 2002

[Lea97] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 1997

[POSA1] F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996

[POSA2] D. Schmidt et al., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley, 2000

[PeSo97] Dorina Petriu and Gurudas Somadder, *A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers,* EuroPLoP 1997

[SAP94] *Introduction to Concepts of the R/3 Basis System*, SAP AG (Basis Modelling Dept.), 1994

[Sore02] Kristian Elof Sørensen, *Session Patterns*, EuroPLoP 2002

[SV96] Douglas C. Schmidt and Steve Vinoski: *Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded Servers (Column 5-7)*. SIGS C++ Report Magazine, Feb.–Aug. 1996.

[Tabe02] Peter Tabeling, *Ein Metamodell zur architekturorientierten Beschreibung komplexer Systeme*, Modellierung 2002, GI-Edition - Lecture Notes in Informatics (LNI) - Proceedings, 2002

[TaGr03] Peter Tabeling, Bernhard Gröne, *Mappings Between Object-oriented Technology and Architecture-based Models*, International Conference on Software Engineering Research and Practice, Las Vegas, 2003

[UML] G. Booch et al., *The Unified Modeling Language User Guide*, Addison Wesley, 1998

[VKZ02] Markus Völter, Michael Kircher, Uwe Zdun, *Object-Oriented Remoting – Basic Infrastructure Patterns*, VikingPLoP 2002

[VSW02] M. Völter, A. Schmid, E. Wolff, *Server Component Patterns*, Wiley, 2002