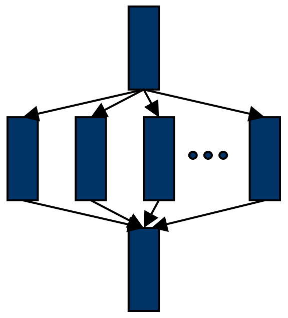
Chapter 6

Client-Server Structures



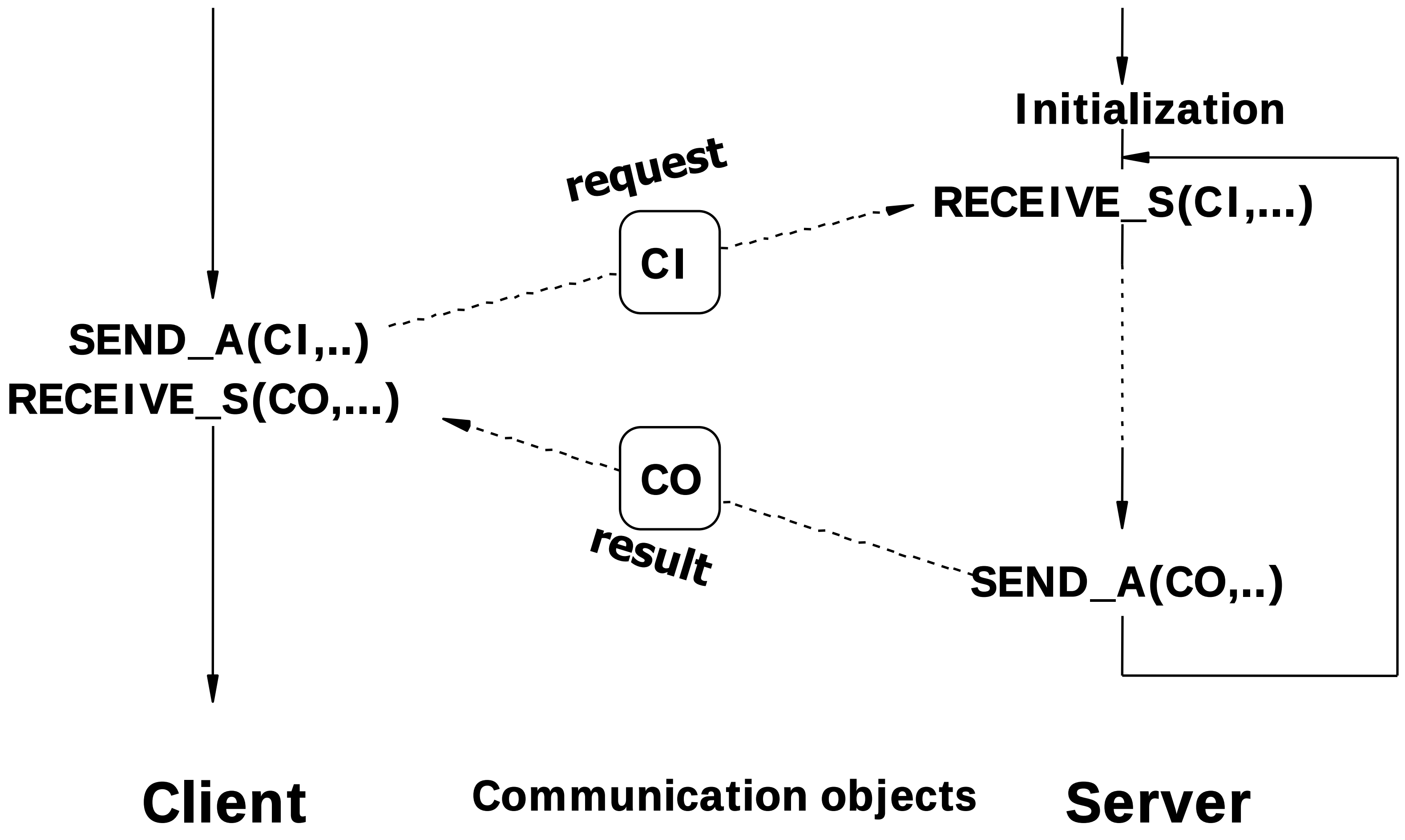
6.1 Client-Server Paradigm

- The fundamental structural element in software systems (centralized and decentralized) is the client-server relation.
- The whole system is decomposed into functional units (*servers*) that deliver some *service*.
- A service consists of one ore more functions (operations, methods etc.) that can be called or requested.
- A server is usually implemented as a process (or thread or group of threads).
- The services of a server can be used by other processes (or threads). They are called *clients*.
- A process is usually server (i.e. offering a service) as well as client (i.e. using other services).
- A complex software system is therefore represented as a network of service relations.

How it works

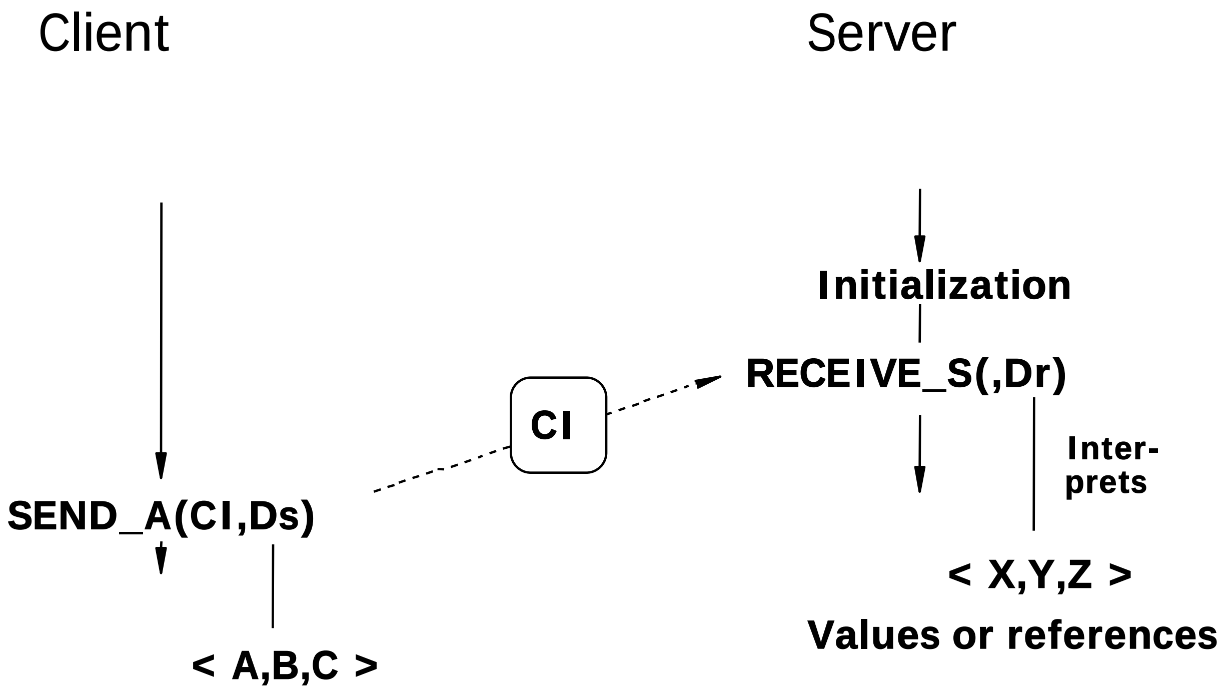
- Looking at a individual service relation two threads are involved.
- The client sends a request to a server.
- The server accepts the requests, processes it and sends a result back to the client. Then it waits for the next request.
- Thus, the server is a cyclic *thread*.
- The communication between client and server takes place by using dedicated communication objects (channels or ports).
- Usually, we have two channels:
  - An input channel at which the server takes the requests
  - A response channel at which the client receives the result.

The Service relation



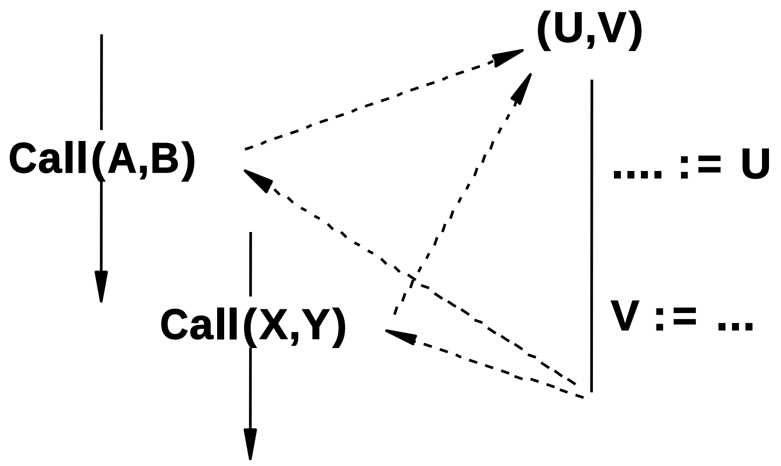
Parameters

The parameters for the request are packed into a message that has to interpreted accordingly (protocol).

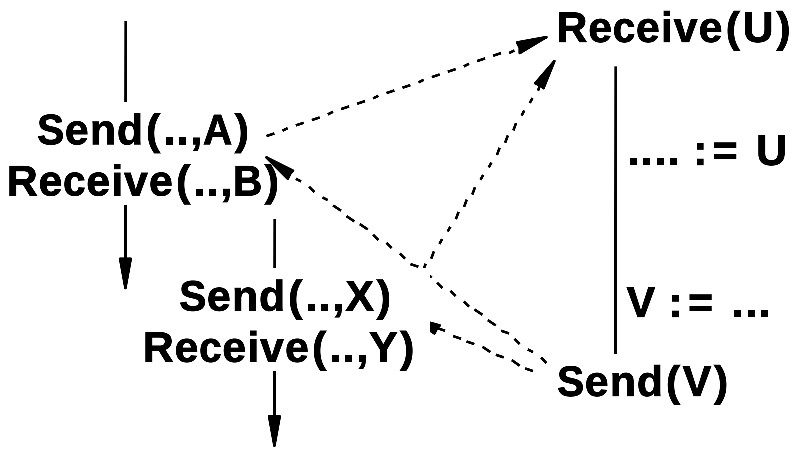


Comparison

The client server relation strongly resembles the conventional procedure call with formal and actual input and output parameters.



Procedure call



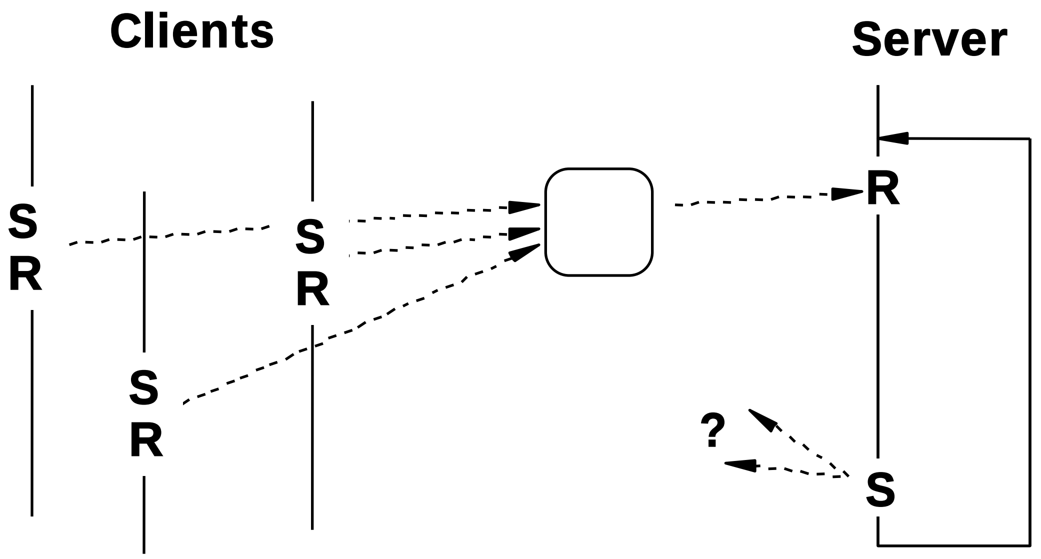
Service request

Return channel

A service is usually used by many clients.

All clients can use the same input channel to submit their requests.

However, if there is only one output or return channel, the result messages cannot be assigned to the clients in a unique way.

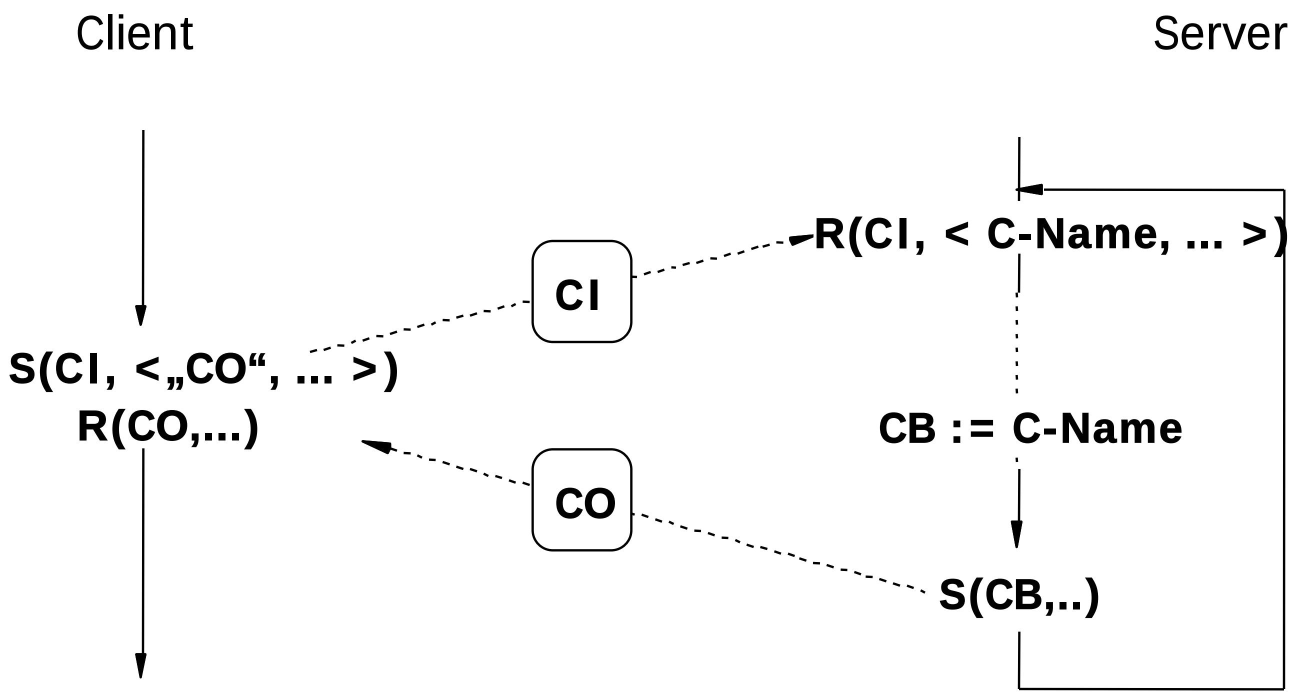


Return channel

Solution:

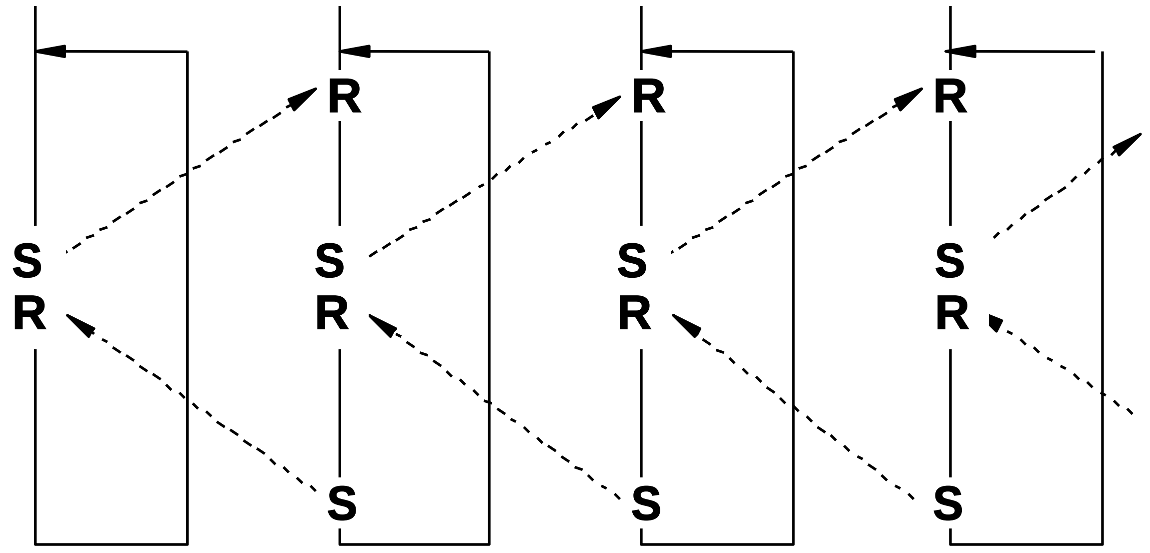
The client tells the server as part of the request message, at which channel it is expecting the result message ("delivery address")

By that, results are delivered correctly.



Service hierarchy

Since server use services of other servers, in many cases we have a dynamic multistage service hierarchy.



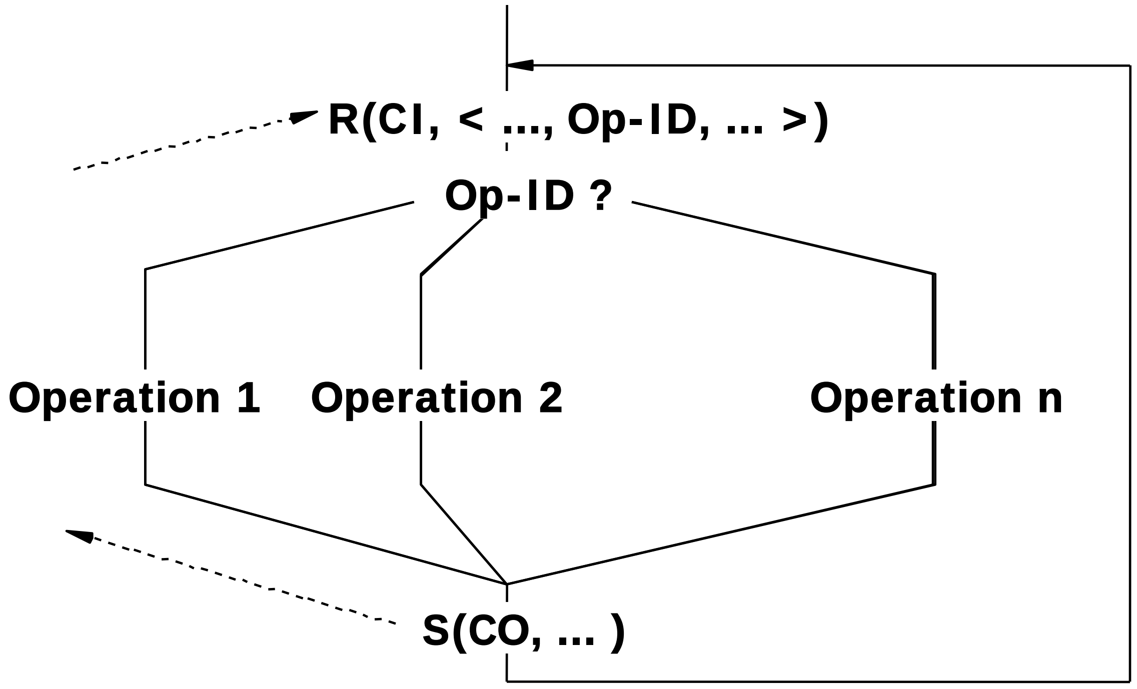
Supporting several operations

- In many cases a server offers several operations that can be called by the client.
- A request corresponds to the execution of one of these operations.
- Example:

Memory management:	allocate, release
File system:	open, close, read, write
Name service:	resolve, insert, delete

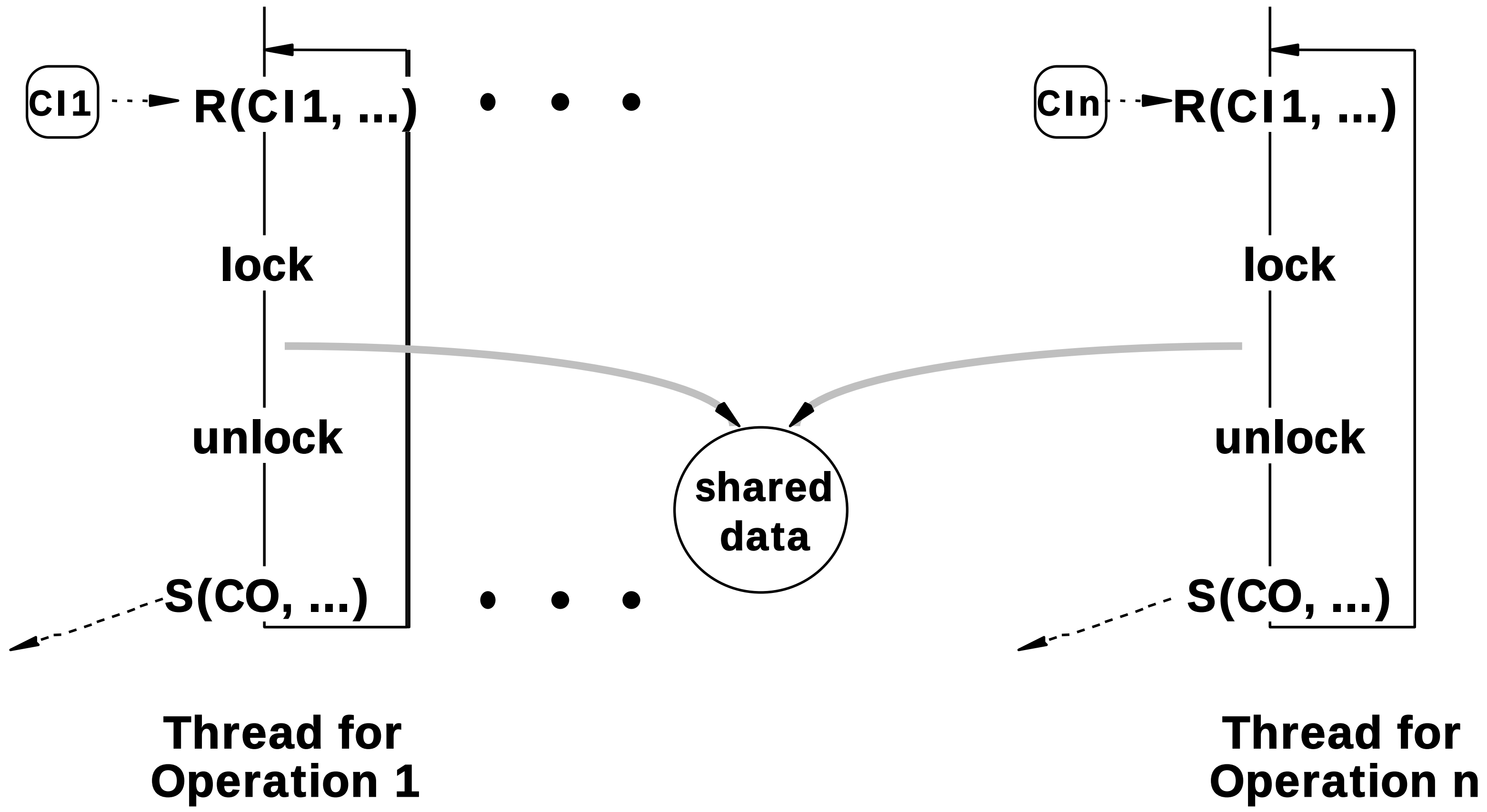
- We can realize these operations within a thread (*Secretary*) or provide for each operation an individual thread (*Team*).

Multioperation server ("secretary“)



Depending on Op-ID we branch into one of these operations.

Server as team of threads



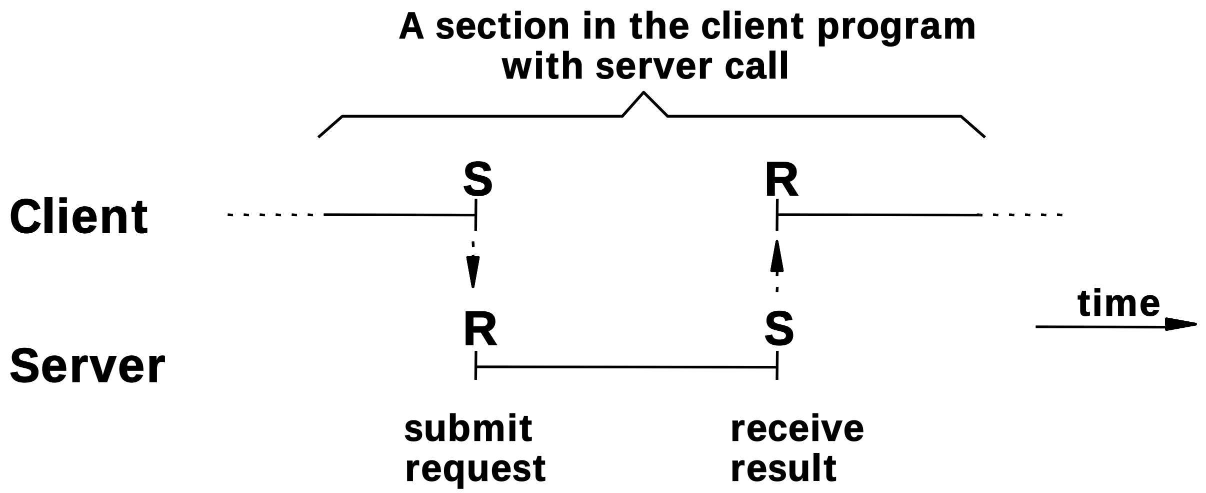
Each thread is responsible for a specific operation and owns an individual entry channel (port).

The selection of the operations is done by selecting the channel (port).

6.2 Parallelism in the service relation

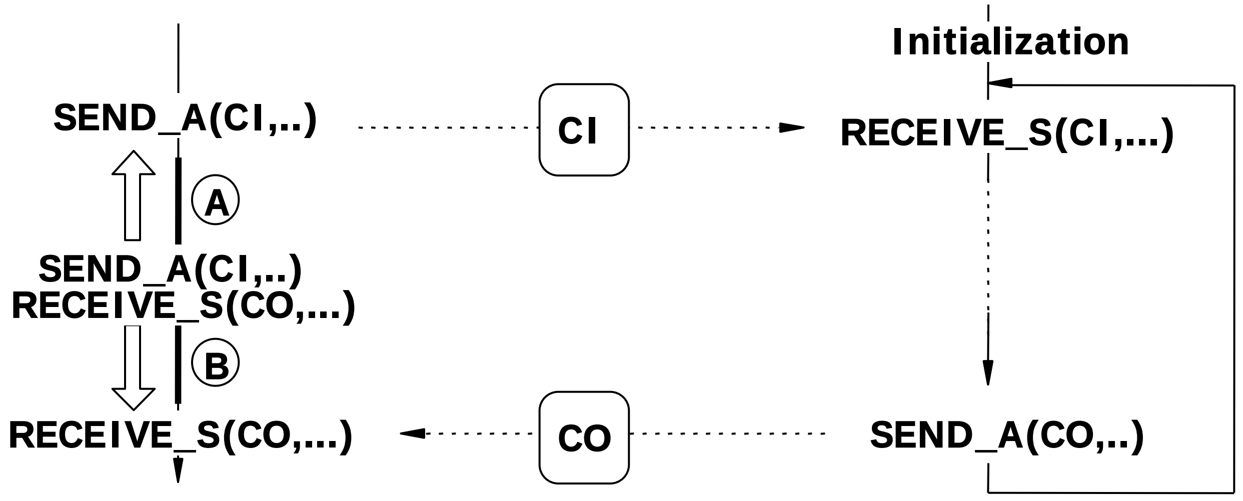
6.2.1 Parallelism between client and server

The service relation as discussed corresponds to the following time diagram:

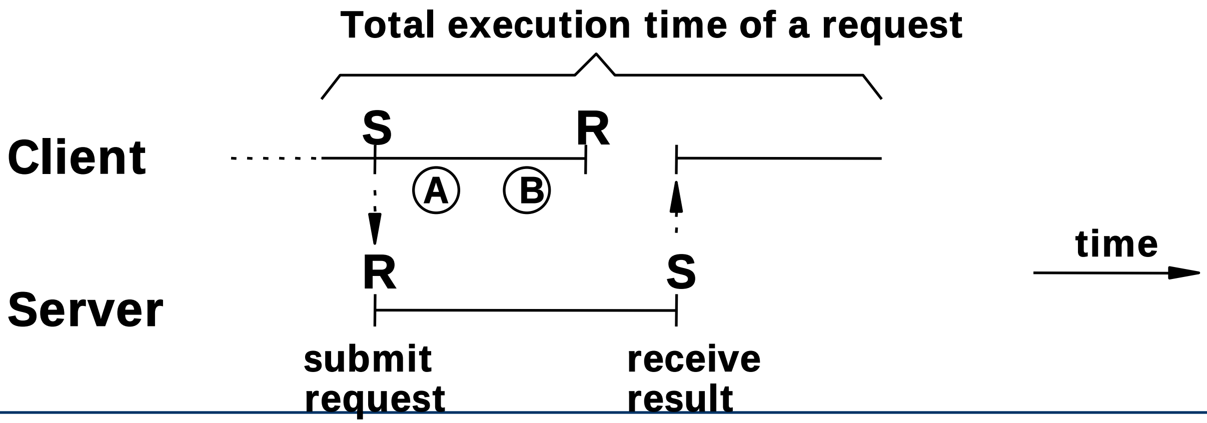


If we submit the request at the earliest point of time and take the result at the latest point of time, we achieve an overlap between client and server activity:

Drift of communication operations



The send operation in the client program has to drift backward, the receive operation forward in the program code.



Conditions for Drifting

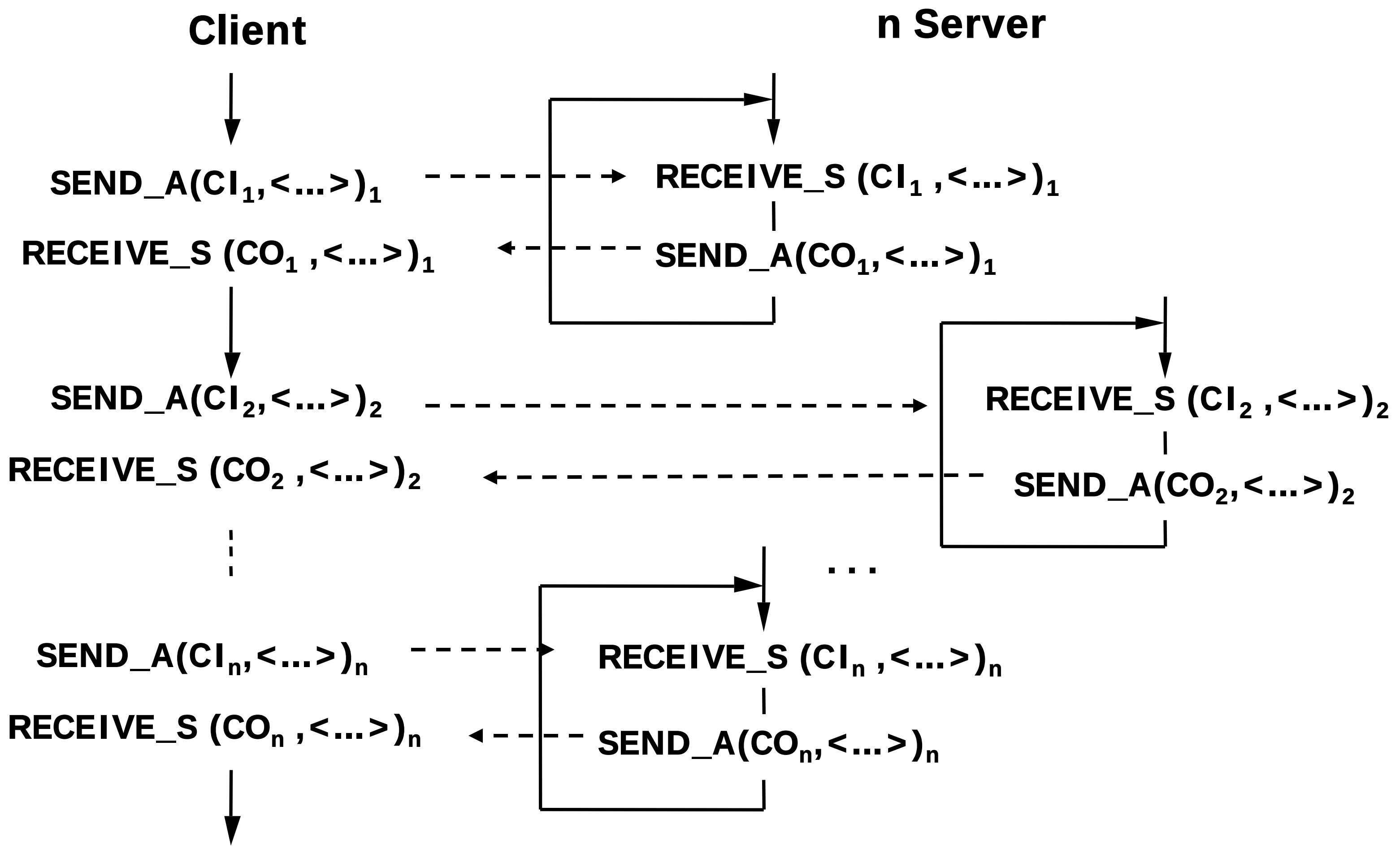
Drift of the communication operation can only be done if no data dependencies are violated. Let be

A	the program section across which the send operation drifts backward
B	the program section across which the receive operation drifts forward
R_S	the set input parameters of the send operation
W_R	the set output parameters of the receive operation
W_A	the set of variables written in section A
R_B	the set of variables read in section B

Then the following must hold:

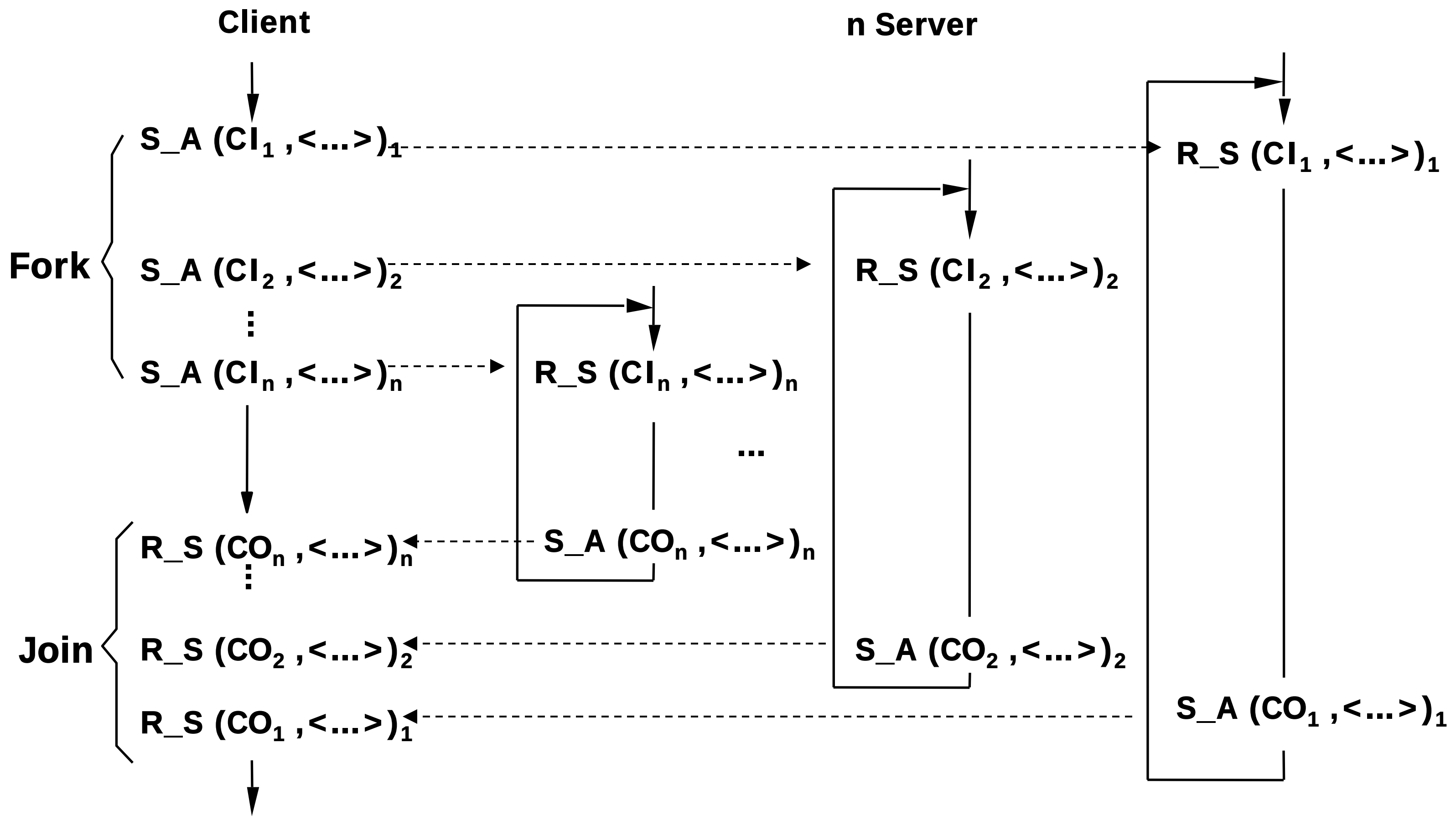
$W_A \cap R_S = \emptyset$:	No variable written in A must be sent.
$(R_B \cup W_B) \cap W_R = \emptyset$:	No variable read or written in B must be received.

Parallelism with multiple requests


Client
n Server
SEND_A(CI$_1$,<...>)$_1$
RECEIVE_S (CI$_1$,<...>)$_1$
RECEIVE_S (CO$_1$,<...>)$_1$
SEND_A(CO$_1$,<...>)$_1$
SEND_A(CI$_2$,<...>)$_2$
RECEIVE_S (CI$_2$,<...>)$_2$
RECEIVE_S (CO$_2$,<...>)$_2$
SEND_A(CO$_2$,<...>)$_2$
...
SEND_A(CI$_n$,<...>)$_n$
RECEIVE_S (CI$_n$,<...>)$_n$
RECEIVE_S (CO$_n$,<...>)$_n$
SEND_A(CO$_n$,<...>)$_n$

A client sends requests to different servers one after another.

Parallelization: Fork/Join-Principle

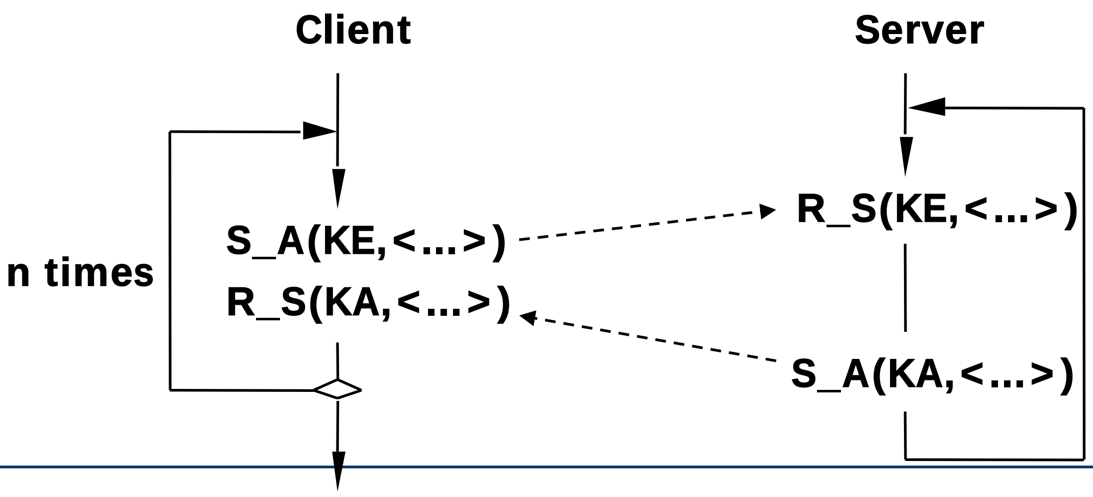


Buffering between client and server

In many cases the service is used not only once or occasionally, but periodically, i.e. in a loop.

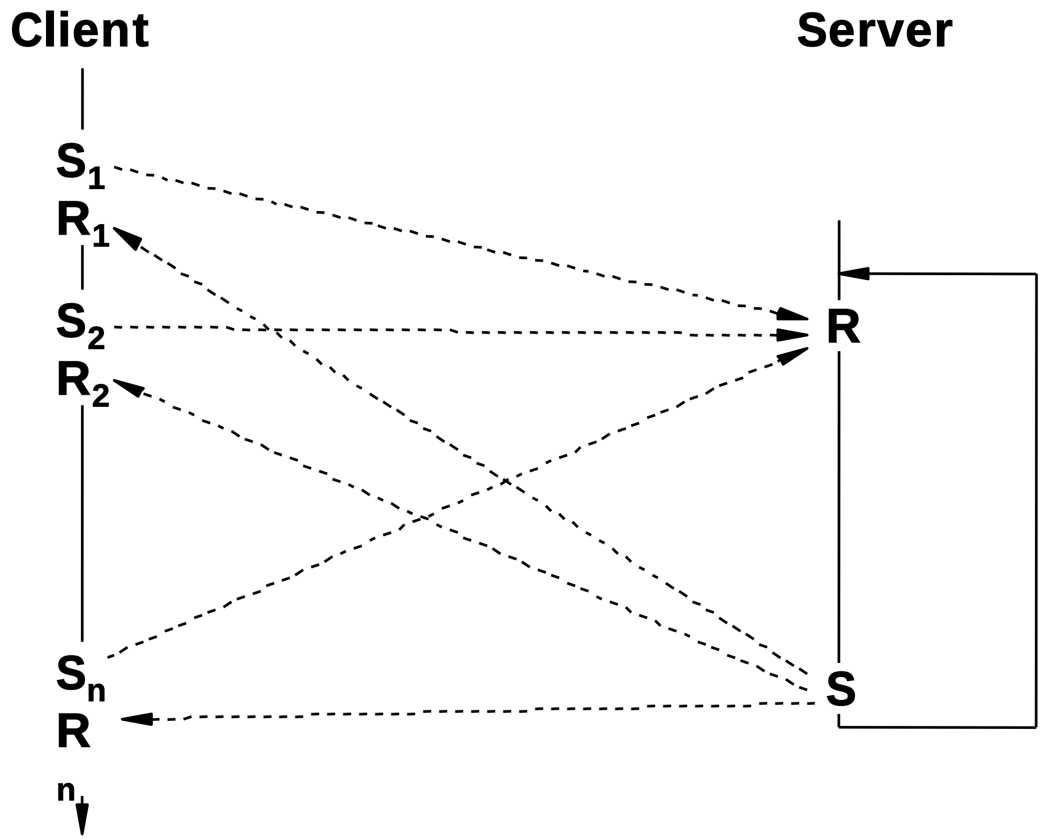
Example: Output of a large amount of data in many small packets:

- Server (e.g. disk driver) can process only blocks of a specific size.
- Client (e.g. file server) must break down the data into small packets and send a request for each packet.



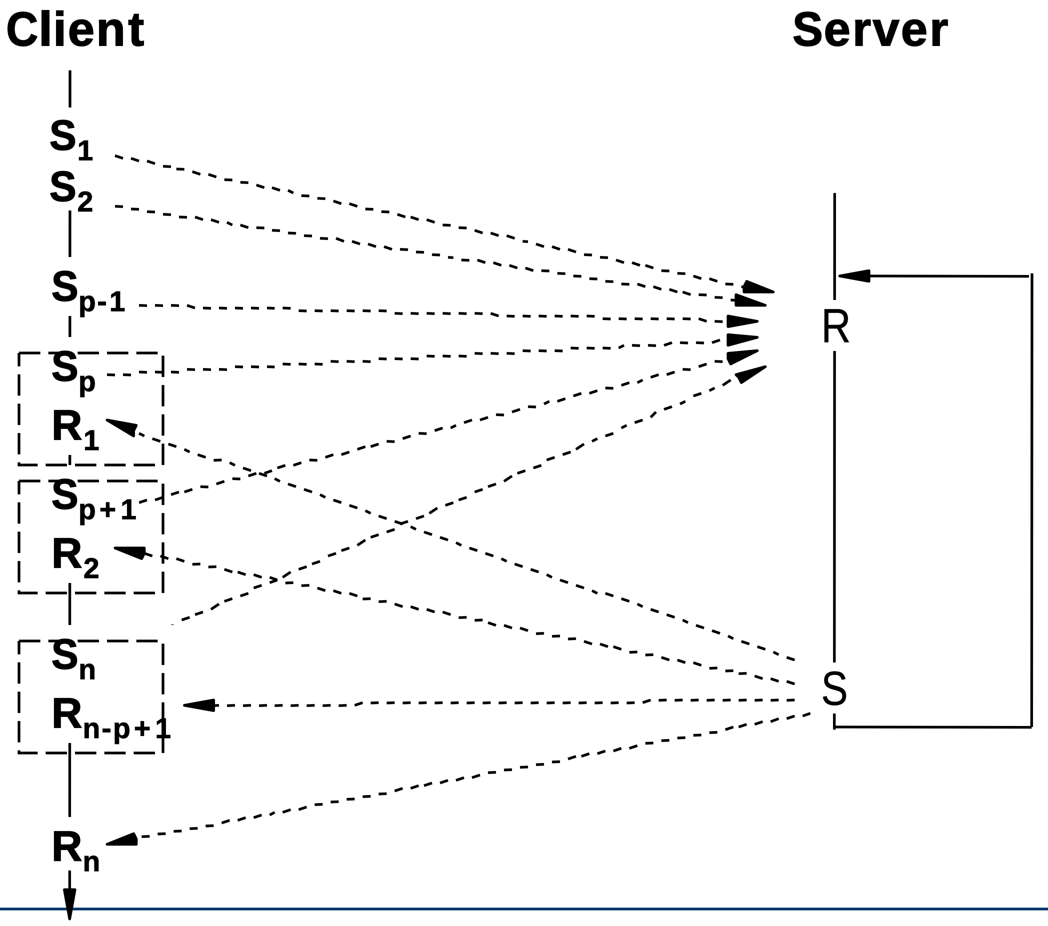
Buffering between Client and Server

Unrolling loop yields:

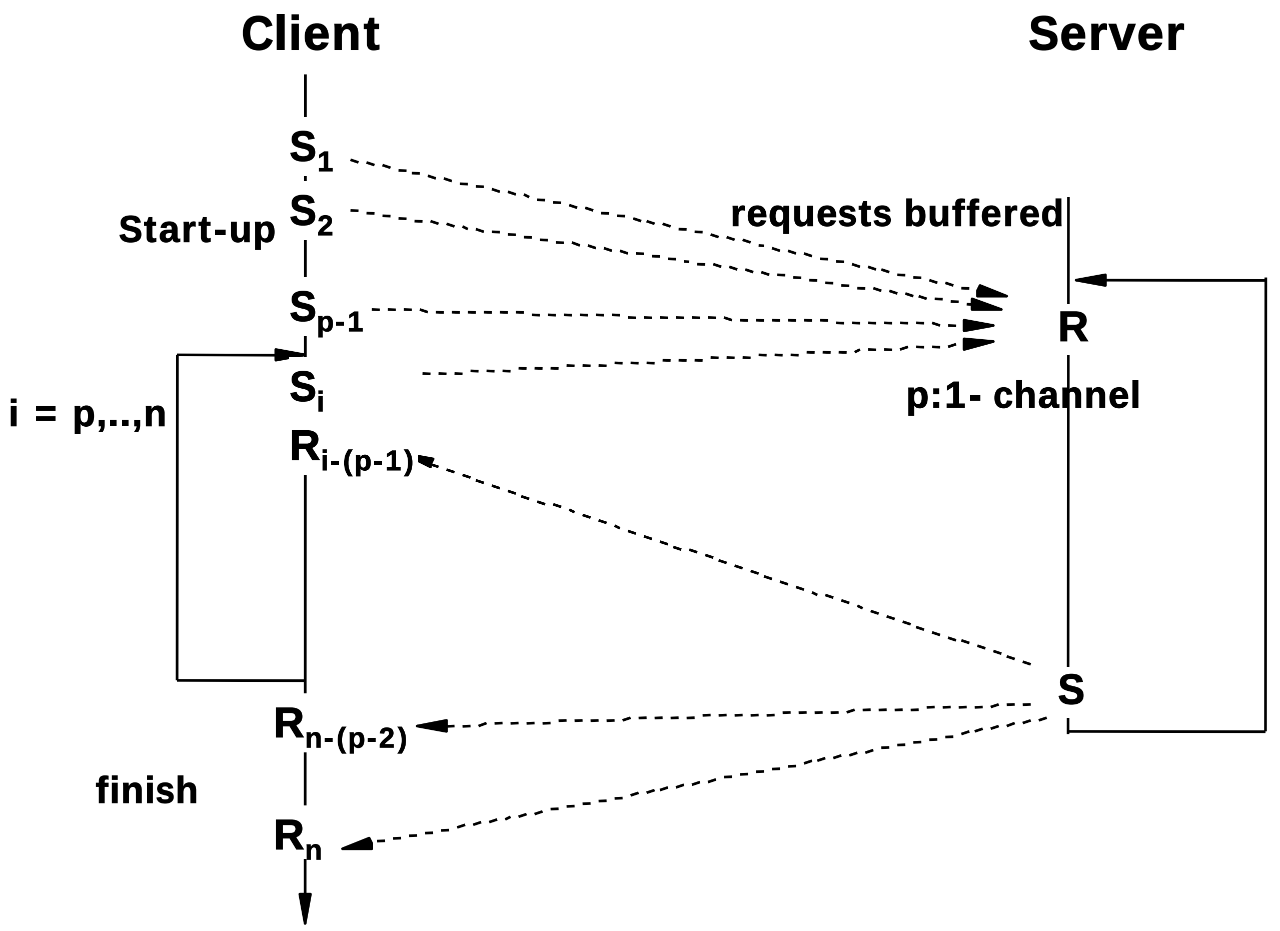


Drift of communication operations

Send operations are pushed forward by *p* -1 positions,
(receive operations backwards by *p* -1 positions)



As loop again (buffering)

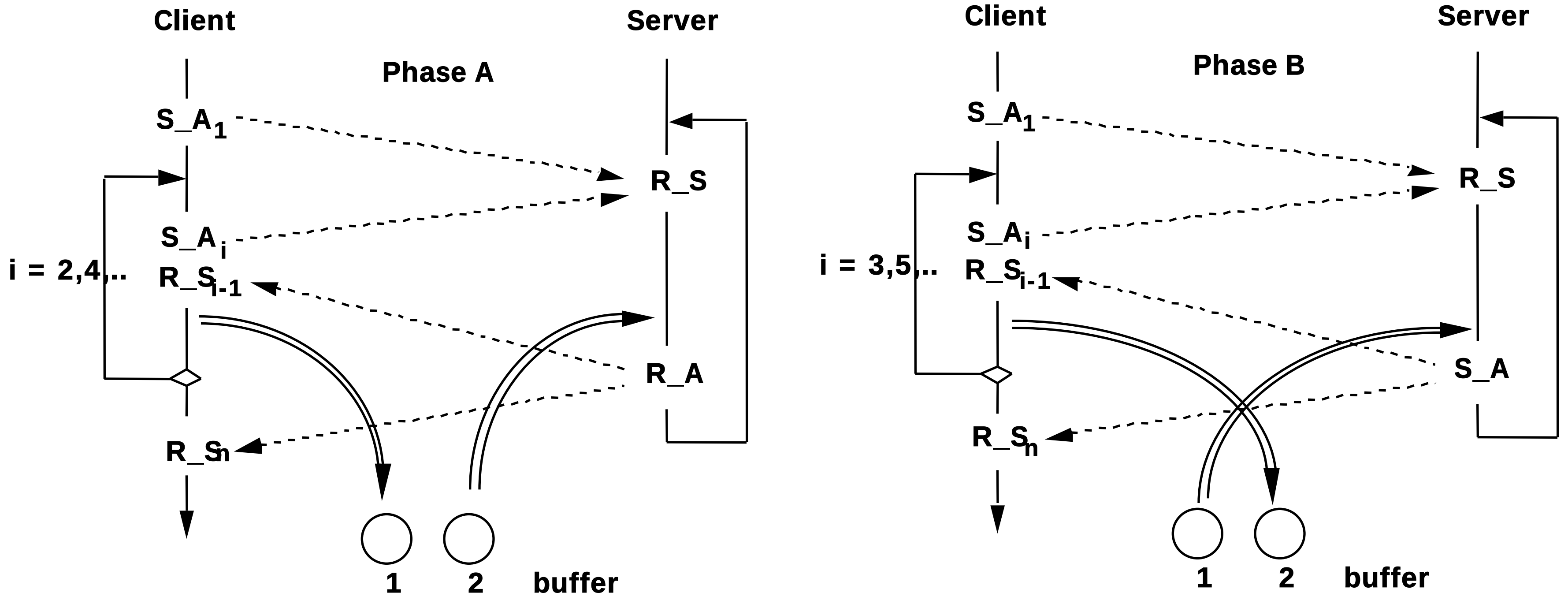


Buffering principle

- Overlapping of client and server activities
- Buffering of requests
- Smoothing differences in service times of requests
- p determines the amount of buffering capability:

  Depending on continuity of request arrivals or service times a suitable *p* can be chosen.
- Widespread usage in software system (OS: input/output, networks: sliding-window protocol)

2-fold buffering



Buffer swapping

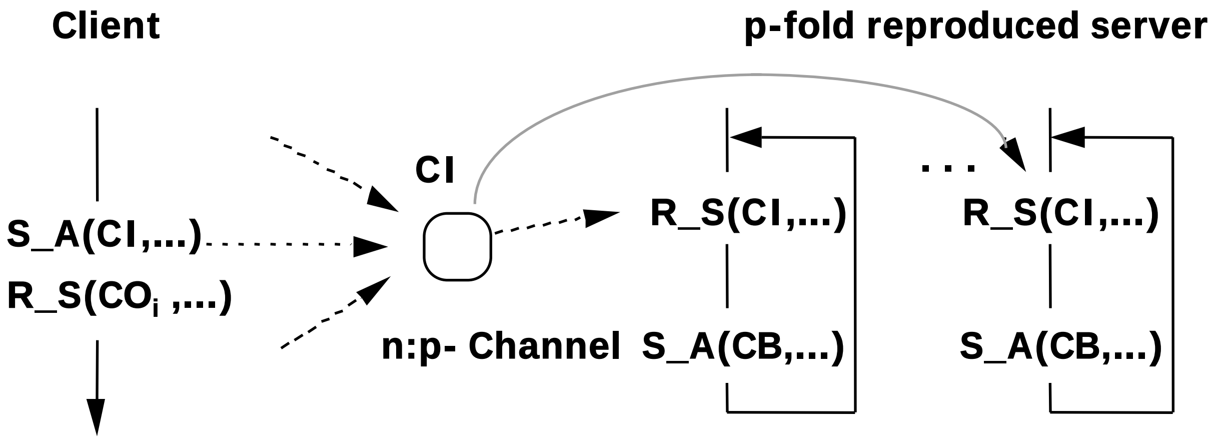
6.2.2 Parallelism within a server

- By overlapping client- and server-activity the *processing time of a request (response time*) can be reduced.
- In addition, we may increase the throughput (requests per time unit).
- This is done by parallelism within the server.
- The server processes many requests simultaneously.

- Mechanisms:
  - Reproduction (Cloning)
  - Pipelining
  - Multiplexing

Reproduction

- The server thread is available as identical copies.
- All these identical threads take requests from a shared input port.
- The client does not see the reproduction.

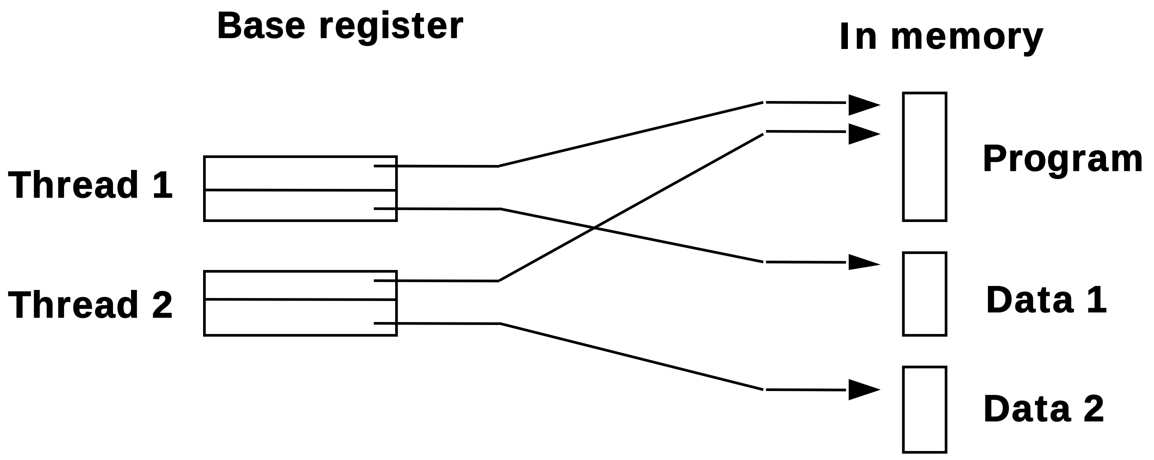


Properties:

- overtaking possible
- up to p requests being processed simultaneously
- easy realization

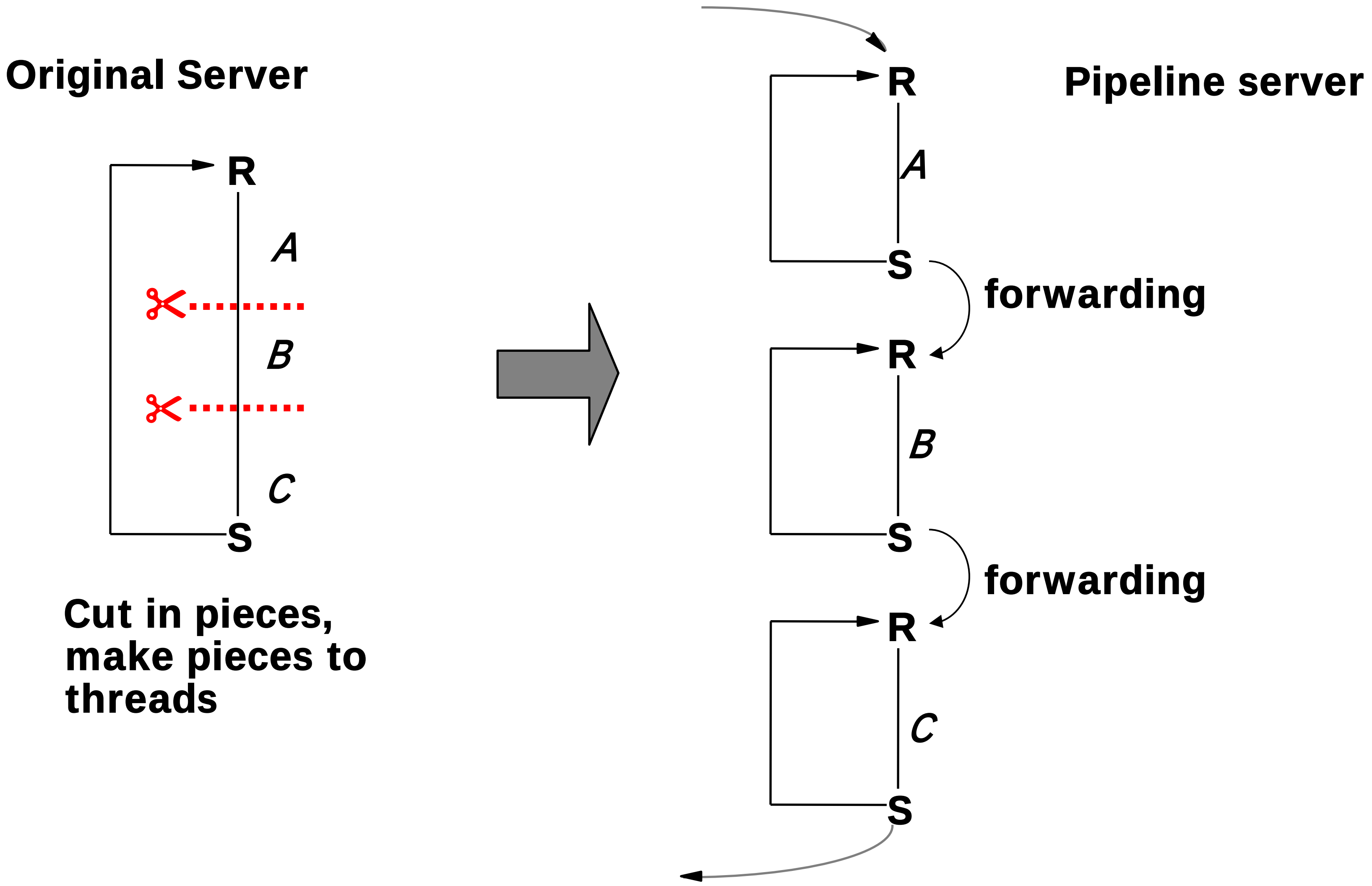
Realization of Reproduction

- Instead of keeping several copies of the program in the memory, it is more economic to make it possible that all identical threads execute the same copy of the program. The program code must be status-free or invariant or reentrant.



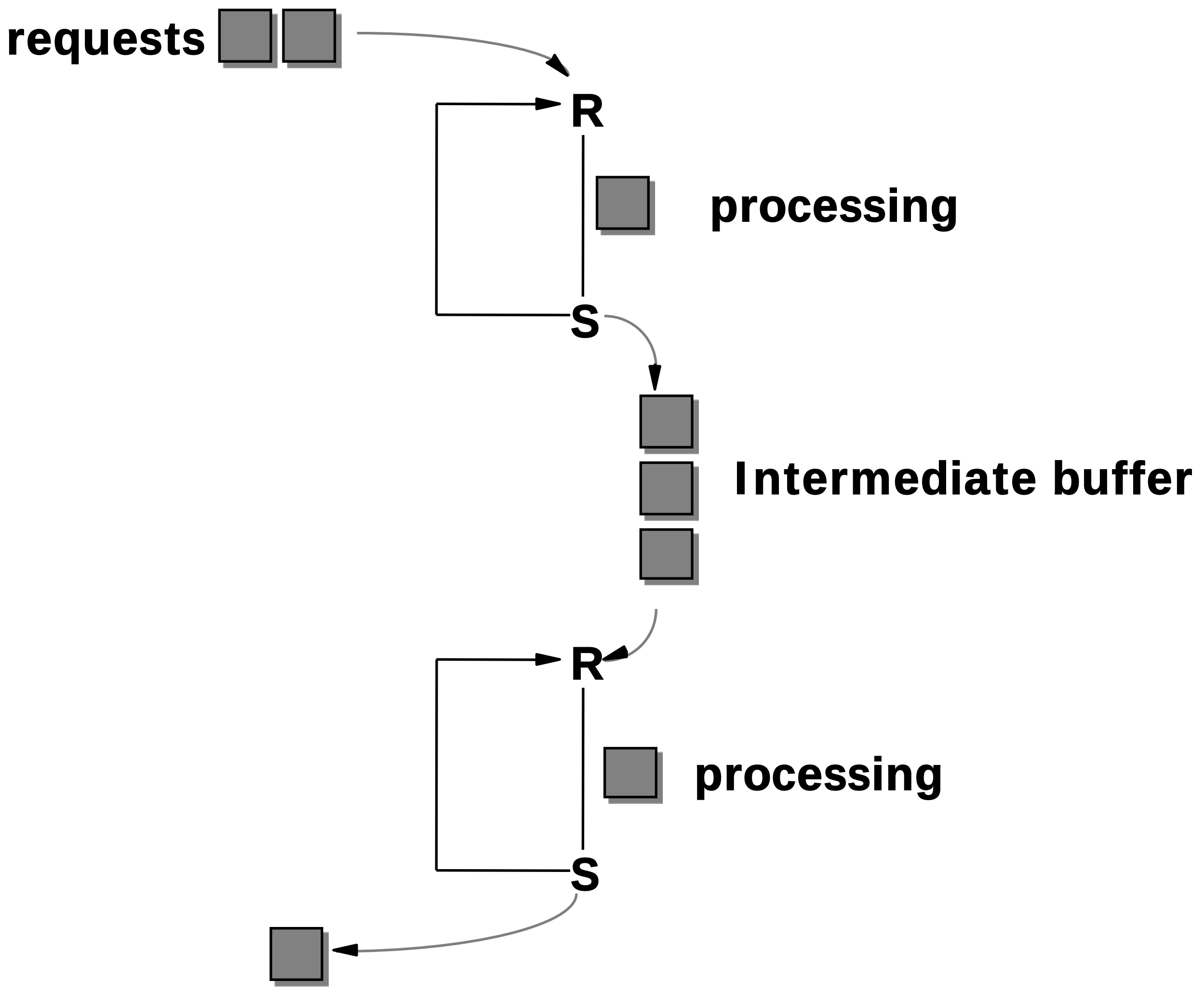
Pipeline (staged server)

Freie Universität Berlin



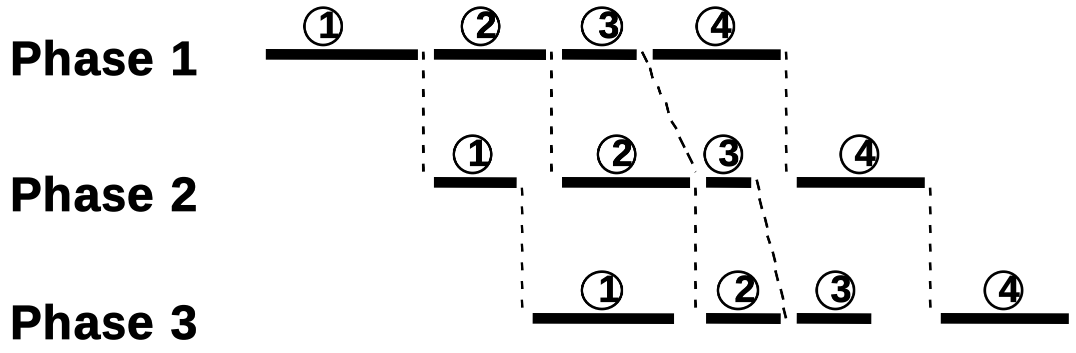
Cut in pieces, make pieces to threads

Request processing in the pipeline



Pipeline

Working principle (with varying service times)

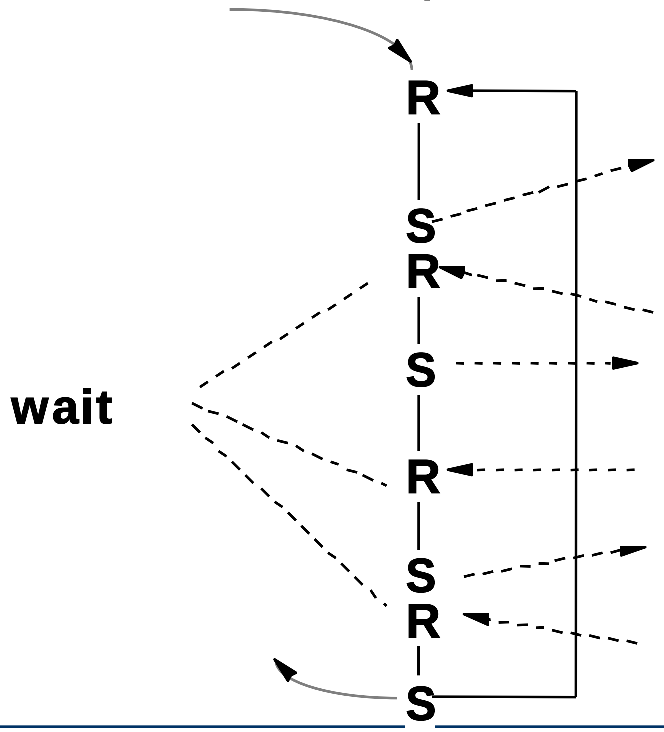


Properties

- Arbitrary number of requests in pipeline
- No overtaking (if internal channels are FIFO, i.e. order preserving)
- Higher transportation overhead (internal channels)
- More difficult to realize

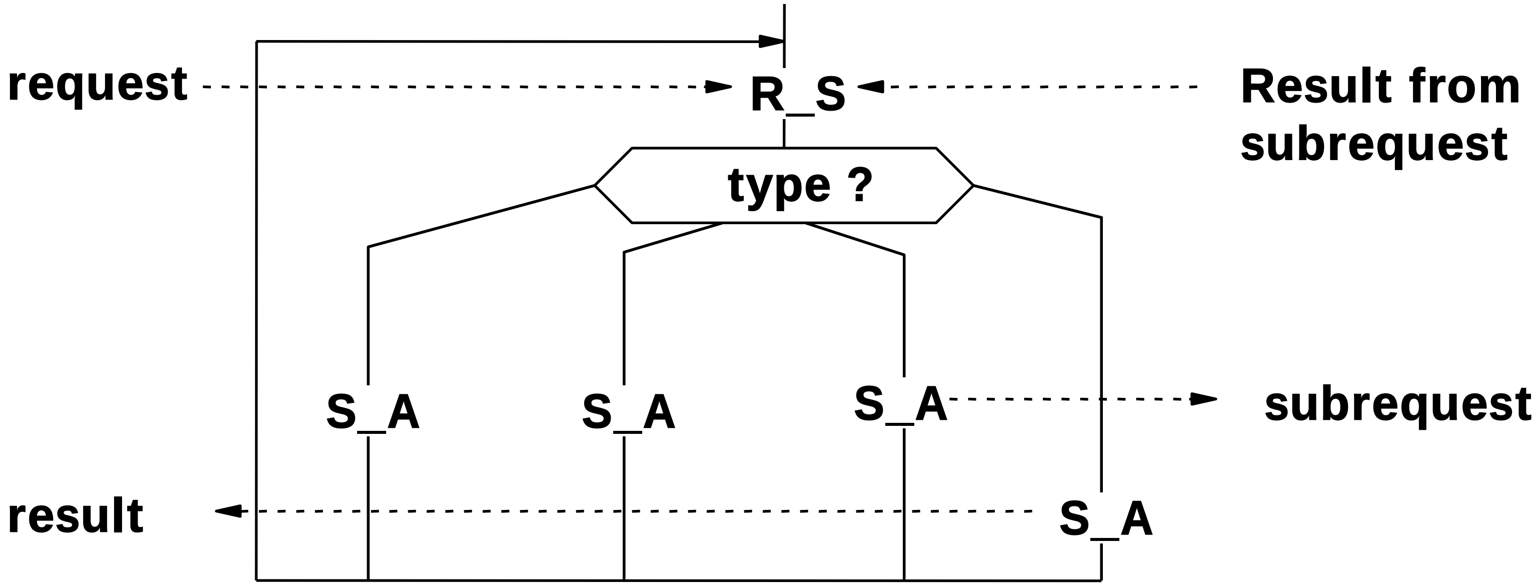
Multiplexing (interleaving, event-programming)

- With complex servers that submit subrequests we may have several waiting positions.
- The thread is stuck at a receive operation waiting for response, although it could continue at another place.
- Example for a structure of a complex service:



Multiplexing

- The thread should continue at that place where work is to be done.
- To wait at a place while at another place work is piling up, is uneconomic.
- To that end it should wait at all receive channels at the same time, to be able to react to all incoming events.
- We therefore combine all channels to one single (super)channel:



Multiplexing

Properties

- Only one thread
- Arbitrary number of requests processed simultaneously
- Difficult to realize

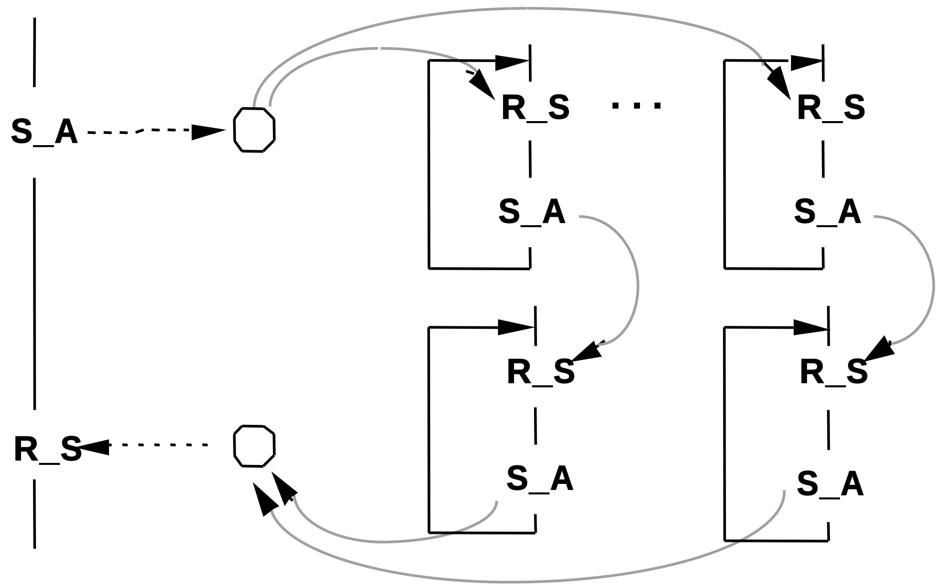
Remark

- A server built according to this multiplexing principle operates (on the software level) in the same way as a processor at the hardware level.
- If a request (thread) cannot be further processed, since it has to wait for something, we simply switch to another request (thread).

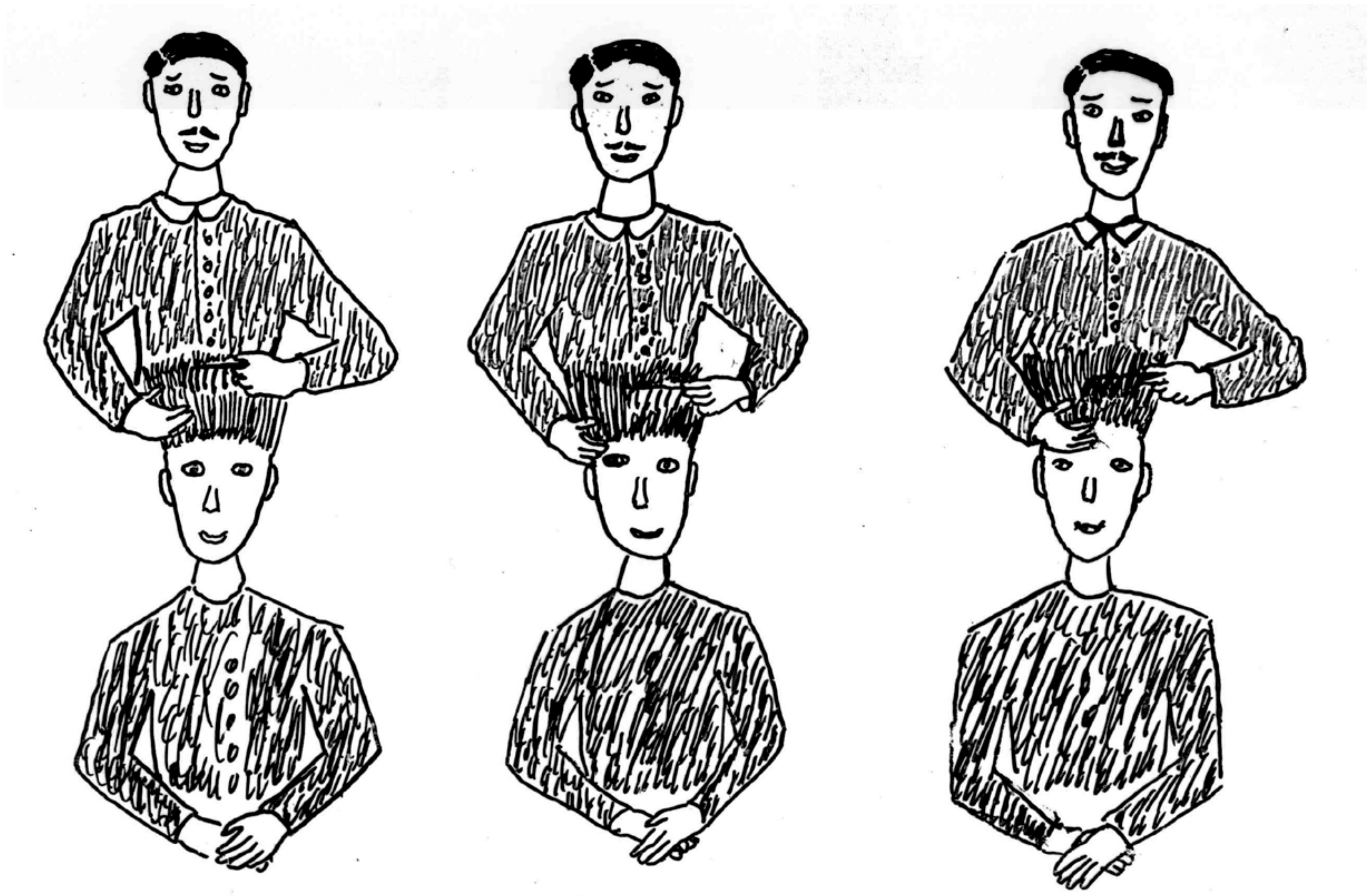
Mix of Parallelism forms

All presented forms of parallelism between client and server or within a server are independent and can be combined.

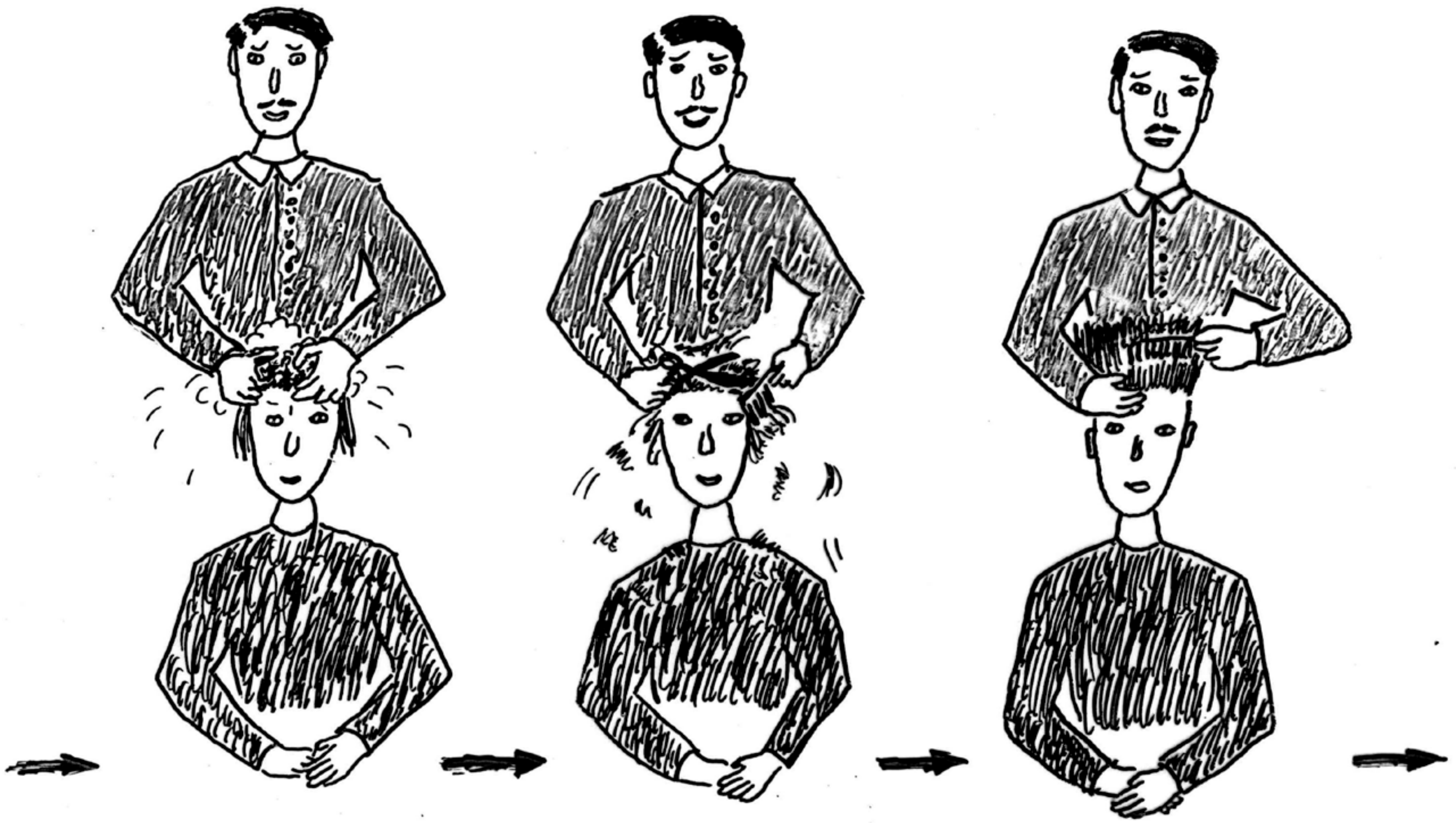
Example:



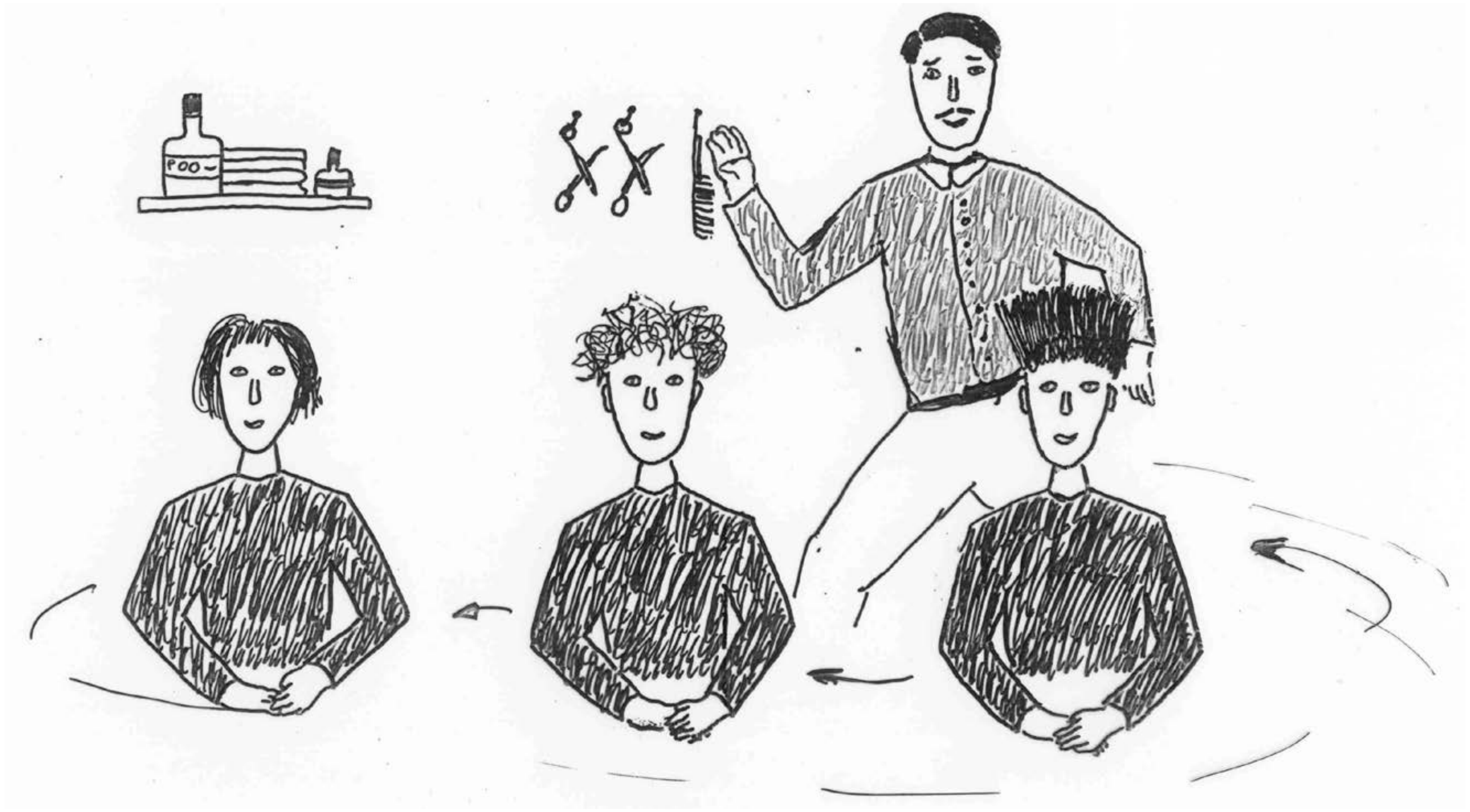
Hair dresser's: Cloning



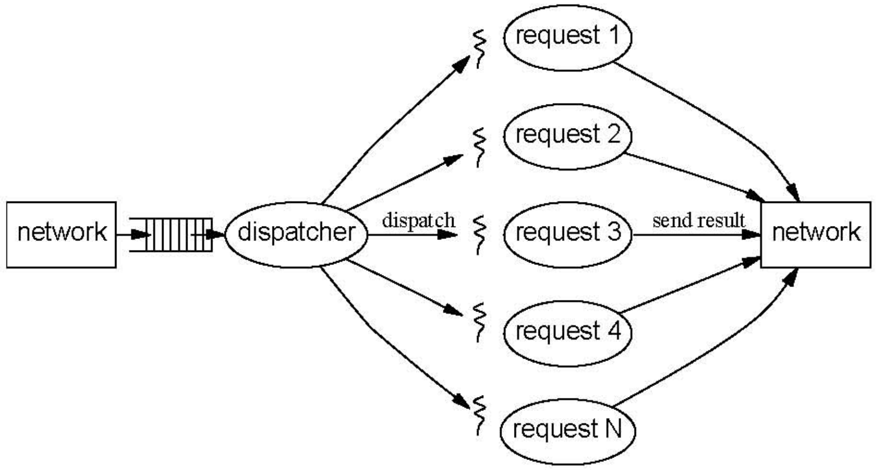
Hair dresser's: Pipelining

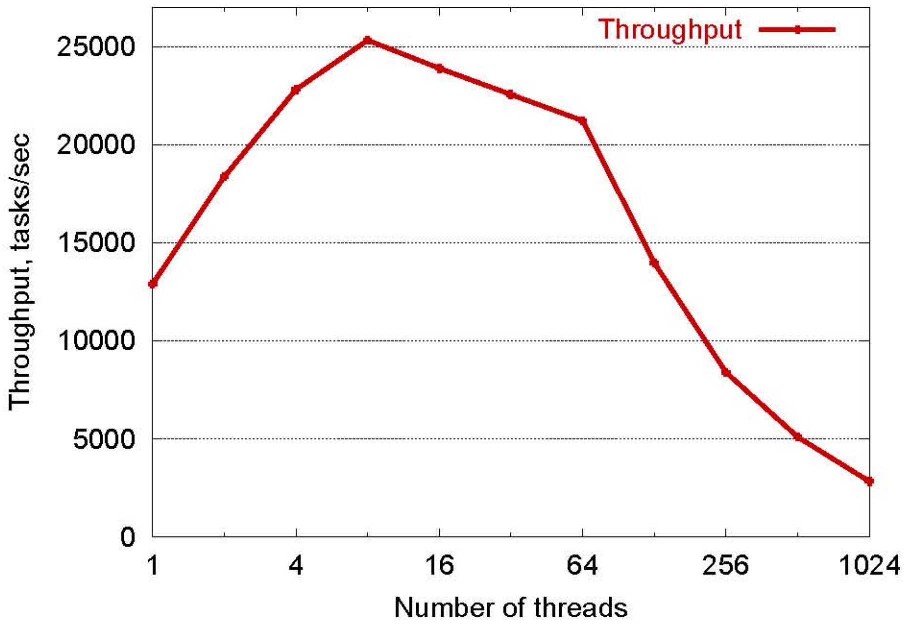


Hair dresser's: Multiplexing



Web Server (thread based concurrency=cloning)





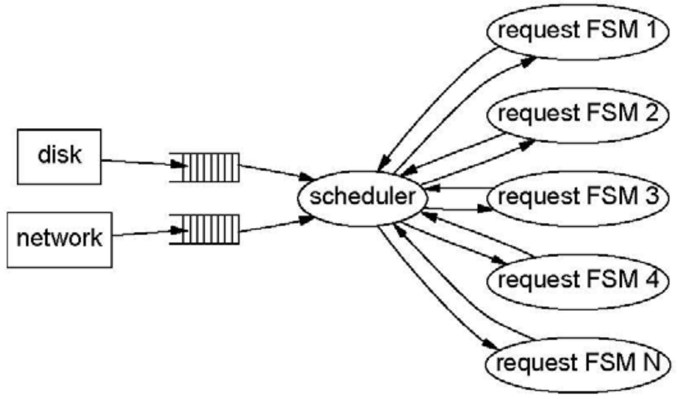
(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)

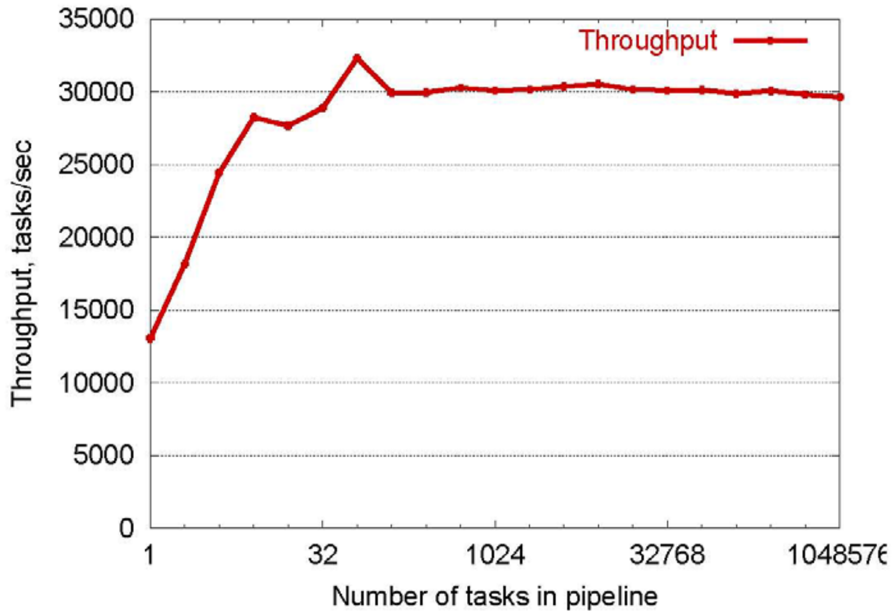
- High resource usage, context switch overhead, contended locks
- Too many threads → throughput meltdown, response time explosion
- Traditional solution: Bound total number of threads
  - ▷ *But, how do you determine the ideal number of threads?*

Matt Welsh, UC Berkeley

Event Driven Concurrency (=multiplexing)

Matt Welsh, UC Berkeley





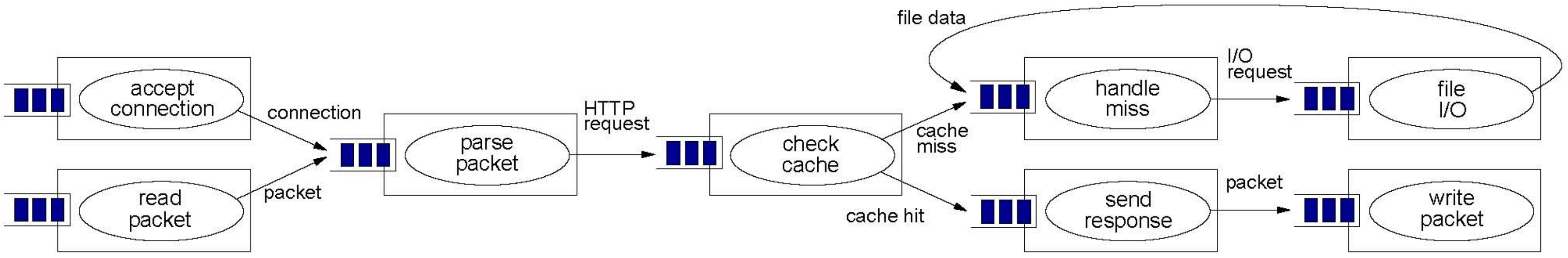
Small number of event-processing threads with many FSMs

- Yields efficent and scalable concurrency
- Many examples: Click router, Flash web server, TP Monitors, etc.

Difficult to engineer, modularize, and tune

- Little OS and tool support: ''roll your own''
- No performance/failure isolation between FSMs
- FSM code can never block (but page faults, garbage collection force a block)

Solution: Staged Server Architecture (=pipelining)



Decompose service into *stages* separated by *queues*

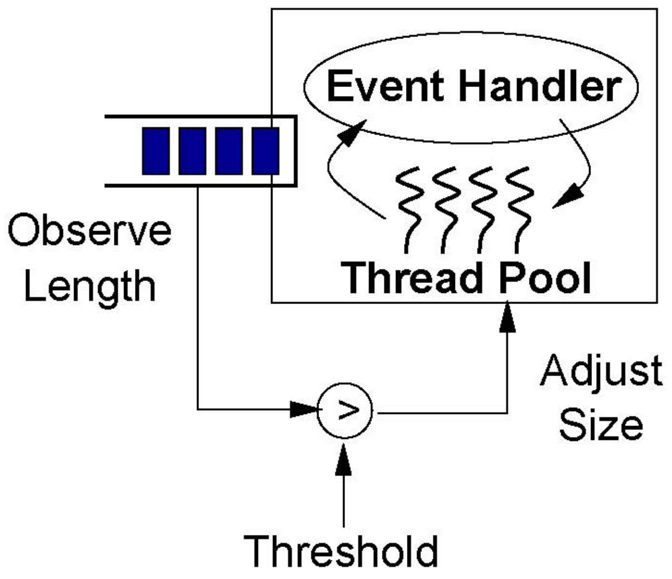
- Each stage performs a subset of request processing
- Stages internally event-driven, typically nonblocking
- Queues introduce execution boundary for isolation and conditioning

Each stage contains a *thread pool* to drive stage execution

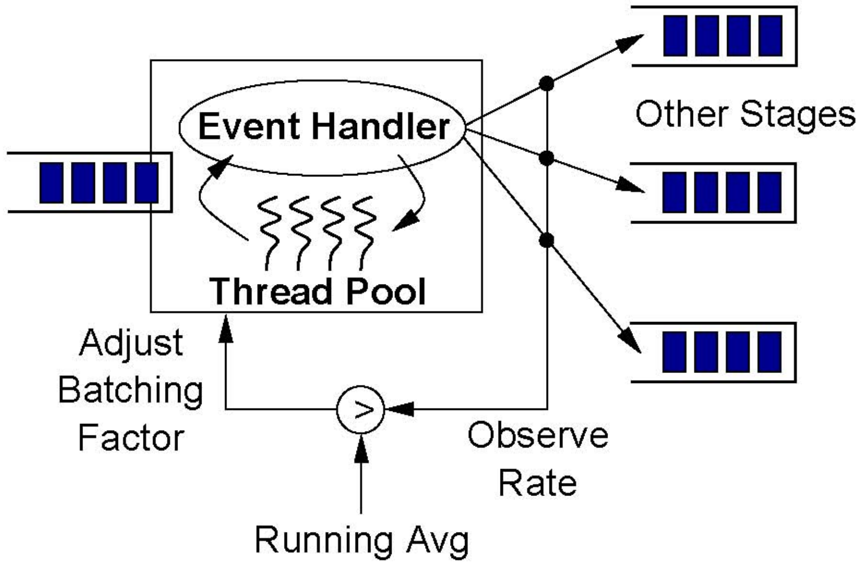
- However, threads are not exposed to applications
- Dynamic control grows/shrinks thread pools with demand

Matt Welsh, UC Berkeley

Components



Control no. of threads



Control no. of requests per thread

Matt Welsh, UC Berkeley

Further Reading

- Wettstein,H.: Systemarchitektur, Hanser, 1993
  Kapitel 10 (in German)
- Welsh,M. et al.: SEDA**:** An Architecture for
  Well-Conditioned, Scalable Internet Services,
  In *Proc. 18th Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.