

Distributed Systems

6 Instances

May-20-2009

Gerd Liefländer

System Architecture Group



Schedule

- Templates for Distributed Instances
 - Process
 - Task with Threads
- Client Templates
- Server Templates
- Stateless and Stateful Client/Server Systems

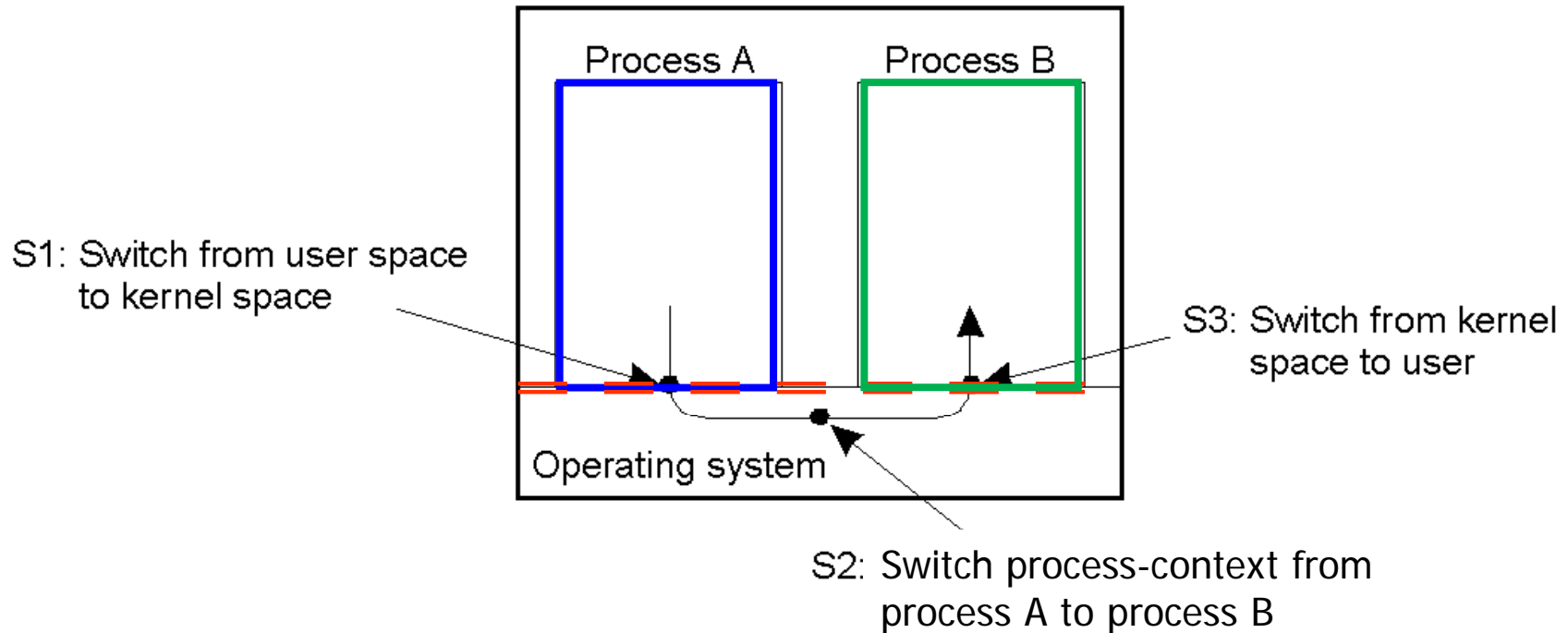


Templates for Distributes Instances

Process
Task
Threads



Usage of Processes in Local Systems



- Collaboration of 2 processes requires cross-AS IPC with additional crossing the border between user-/kernel-space twice and performing a `process_switch` (sooner or later)
- Main drawback of process model: high cost, e.g.
 - Creating and destroying a process
 - Manipulating a process, i.e. sending a IPC



Activity Templates

- To establish high performing DS we **should** use well fitting activity templates, e.g. either
 - Process
 - Task consisting of
 - PULTs
 - KLTs
 - HYBRIDs
- *Why not a mix of processes and tasks?*



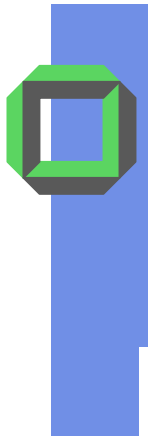
“Passive” Templates

- Remote procedure
- Remote object
 - RMI
 - Single or replicated remote object
- Distributed object
 - Single or even replicated distributed objects

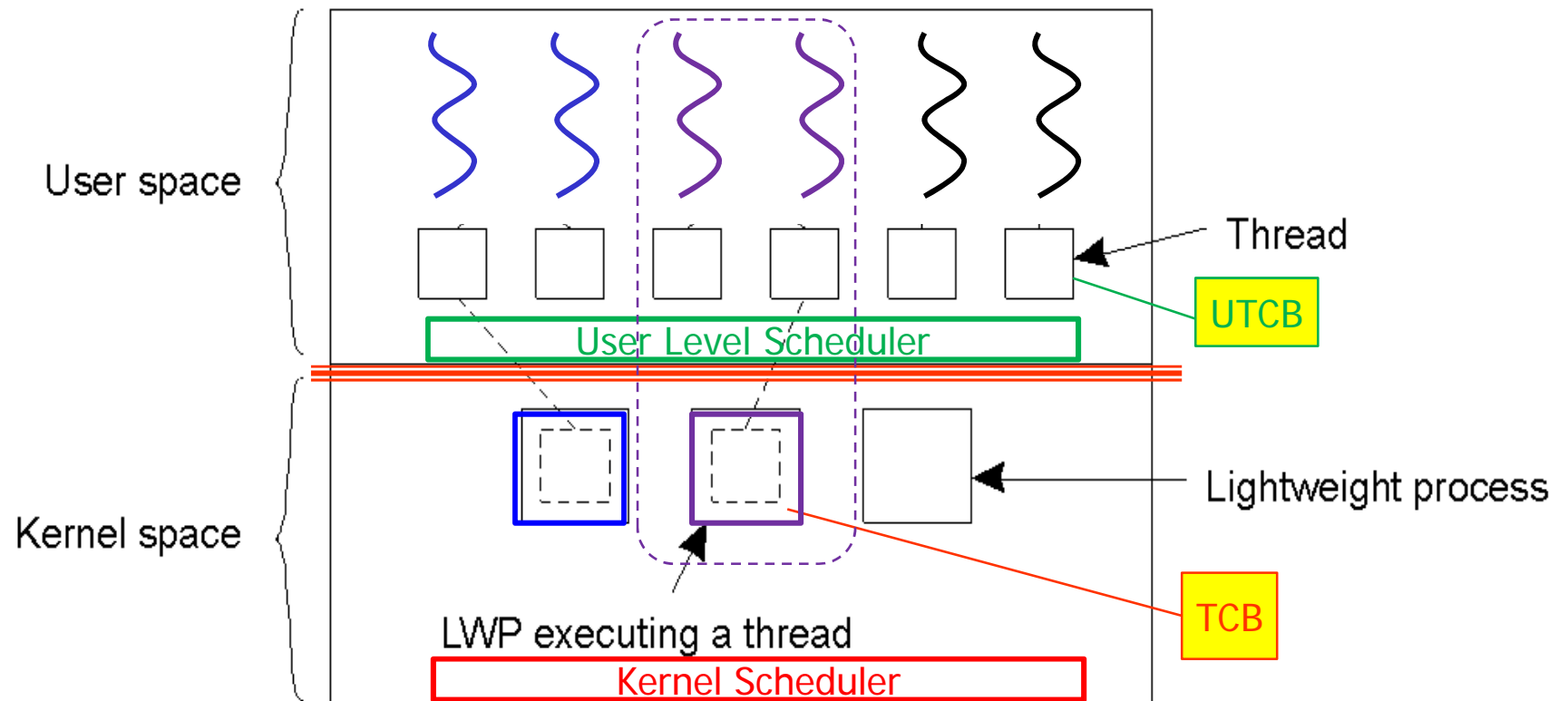


Review: Threads in Local Systems

- A task with only PULTS is blocked completely on a **blocking** system call
- Synchronization and interaction of PULTS can be more efficient, however, there is no real parallelism inside a PULT-task
- A KLT-Task with multiple blocking-system calls can run efficiently on any system
- On a SMP, a KLT-task can use more than one CPU as long as the KLTs do not conflict on global data too often
- Any KLT manipulation is done by the kernel, e.g. high overhead
- Hybrids tasks are more complicated, however can combine the advantages of both pure thread solutions



“Hybrid” Thread Implementation



- Combining KLTs respectively lightweight processes and user-level threads



Thread Paradigms in DS

- Different categories of usage:
 - **Defer (background) work:** additional thread does some work not that vital to the main activity, e.g.
 - printing a document
 - sending mail
 - **Pumps** used in pipelining: use output of a thread as input from the predecessor thread and produce output to be consumed by the successor thread
 - **Sleepers:** threads that repeatedly wait for an event to execute
 - check for **network connectivity** every x seconds



Multithreading Debate

- Three major options:
 - Single-threaded server: only does one thing at a time, uses send/receive system calls and **blocks** while waiting
 - Multi-threaded server: internally concurrent, each request spawns a new thread to handle it
 - Upcalls: event dispatch loop does a procedure call for each incoming event



Multithreaded RPC

- Each incoming request is handled by spawning a new thread
- Designer must implement appropriate mutual exclusion to guard against “race conditions” and other concurrency problems
- Ideally, such a server is more active because it can process new requests while waiting for its own RPC's to complete on other still pending requests

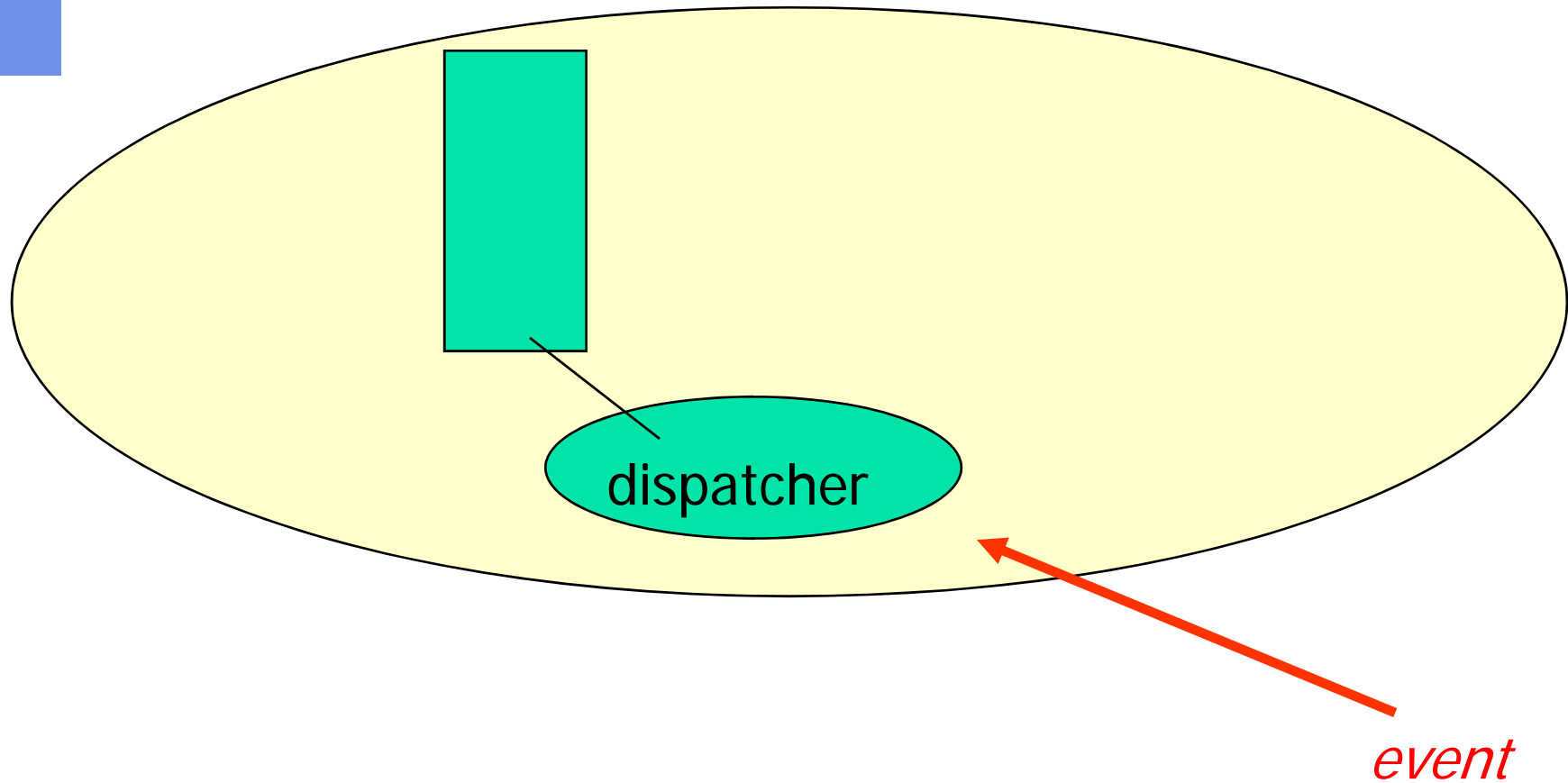


Problems with Multithreading

- Reentrancy can come as an additional requirement, can be an undesired surprise
- Each thread (mini-server) needs a new stack, hence consumption of memory can be too high
- Stacks for threads must be finite and can overflow, corrupting the address space (of the macro-server)
- Concurrency “bugs” are very hard to find due to non-reproducible scheduling orders
- Deadlock remains a risk, now associated with concurrency control

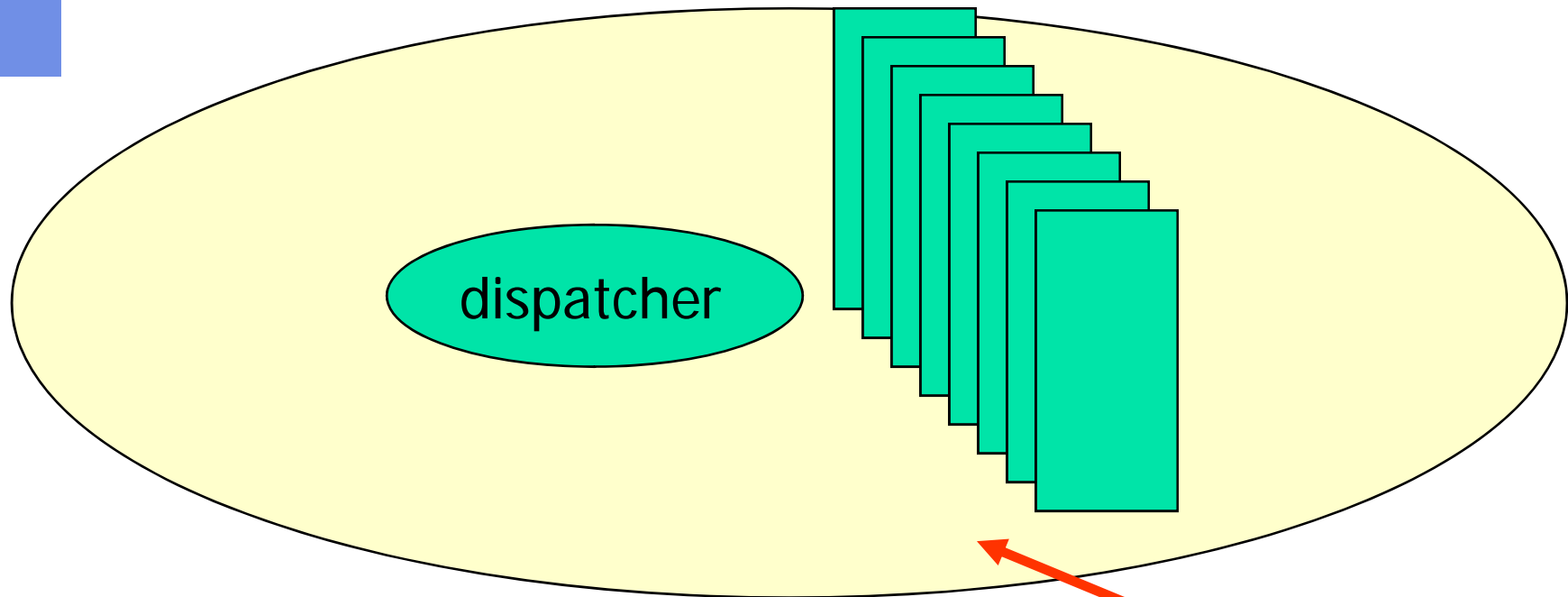


Threads: can spawn too many





Threads: can spawn too many



Eventually, application becomes bloated,
begins to thrash. Performance can drop,
clients might think the server has failed

event



Upcall Model

- Common in windowing systems
- Each incoming “event” is encoded as a small descriptive data structure
- User registers event handling procedures
- Dispatch loop calls the procedures as new events arrive, waits for the call to finish, then dispatches a new event



Upcalls combined with Threads

- Perhaps the best model for RPC programming
- Each handler can be tagged:
 - either needs a (new) thread or
 - can be called
- Developer must still be very careful where threads respectively procedures are used



Usage of Threads in Clients (1)

- Multithreaded clients
 - Main issue is hiding network latency
 - Round trip delay in WANs in *ms* or *s*
- Example: Web Client
 - Web browser scans an incoming HTML page, and finds that more files need to be fetched
 - Each file is fetched by a separate thread, each one doing a (*blocking*) HTTP request
 - As files come in, the browser displays them



Usage of Threads in Clients (2)

Multiple RPCs:

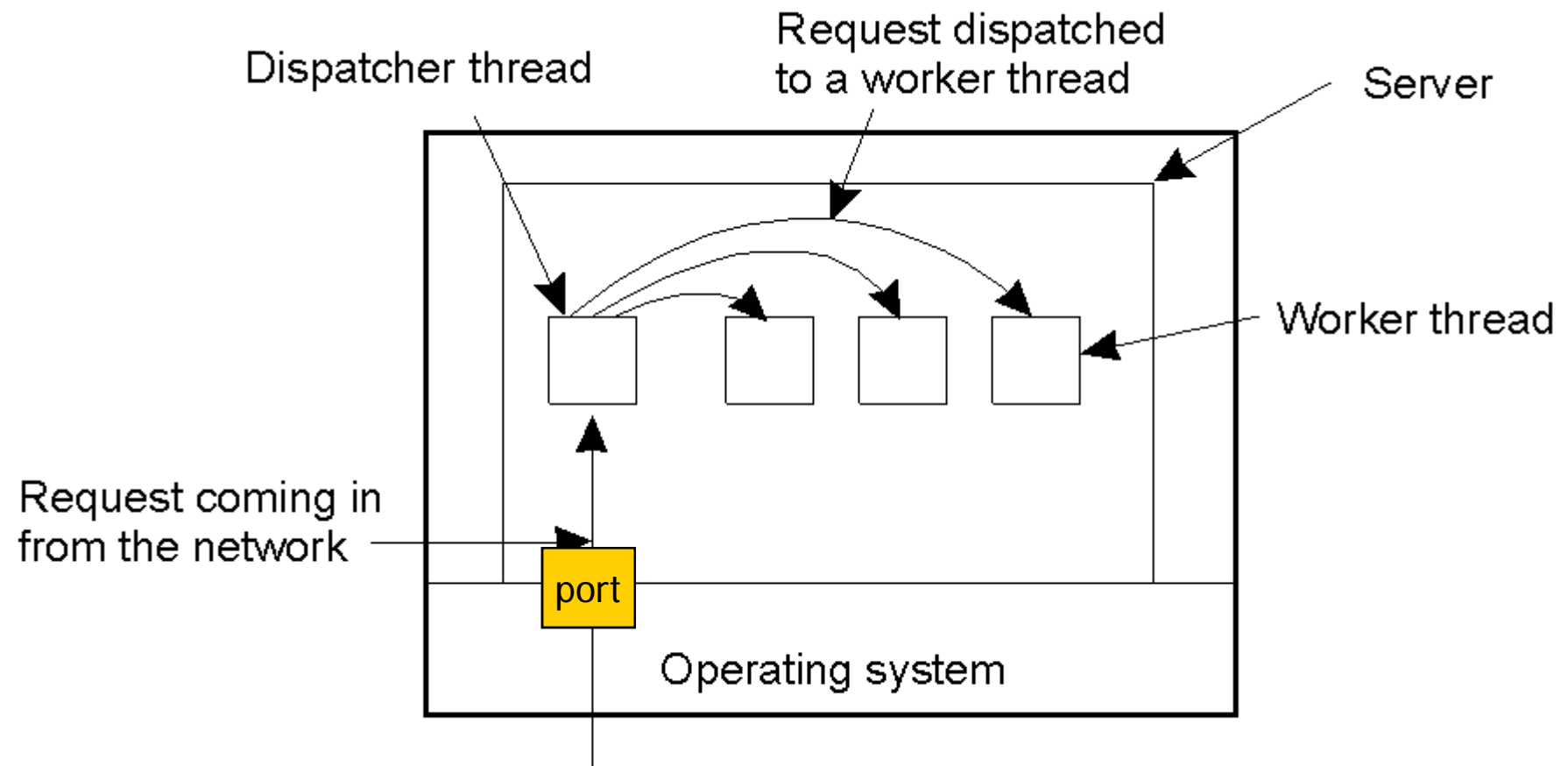
- An application calls several RPCs in a row, each one by a different “communication” thread
- It then waits until all results have been returned
- If the $r > 1$ RPCs can be sent to different servers on different nodes, we can get a **speedup** compared to $r > 1$ RPCs one after the other

Conclusion:

- Divide a huge multi-functional **macro-server** into multiple specialized **mini-servers**
- These $s > 1$ mini-servers still can be mapped to **one AS**



Threads in Servers



- Multithreaded server template: [dispatcher/worker model](#)



Multithreaded File Server

Either a permanent pool of workers or pop-up threads

File server dispatcher

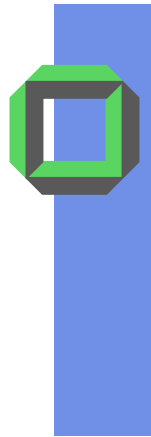
worker threads

Mailbox for File server



File server requests

kernel

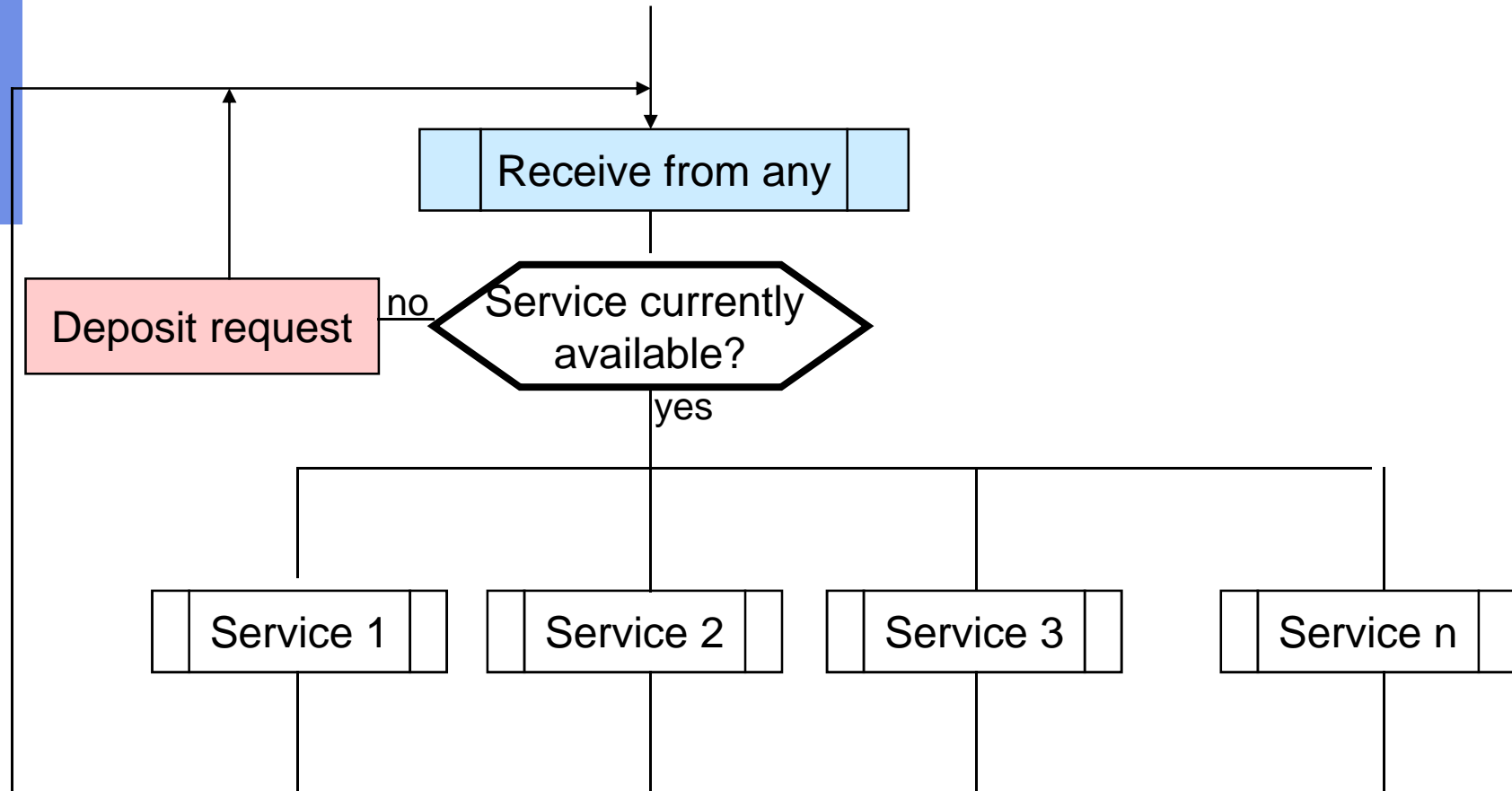


Other Server Implementations

- Team Model
 - No dispatcher thread, every worker thread can receive request messages from a common mail box
- Pipelined Model
 - Usable if different stages within the handling of a request can be distinguished
- Finite-State Machine (Event Driven Server)
 - Dependent on incoming message file-server executes different subprograms, not allowed to contain non-blocking system-calls, otherwise complete server would be blocked



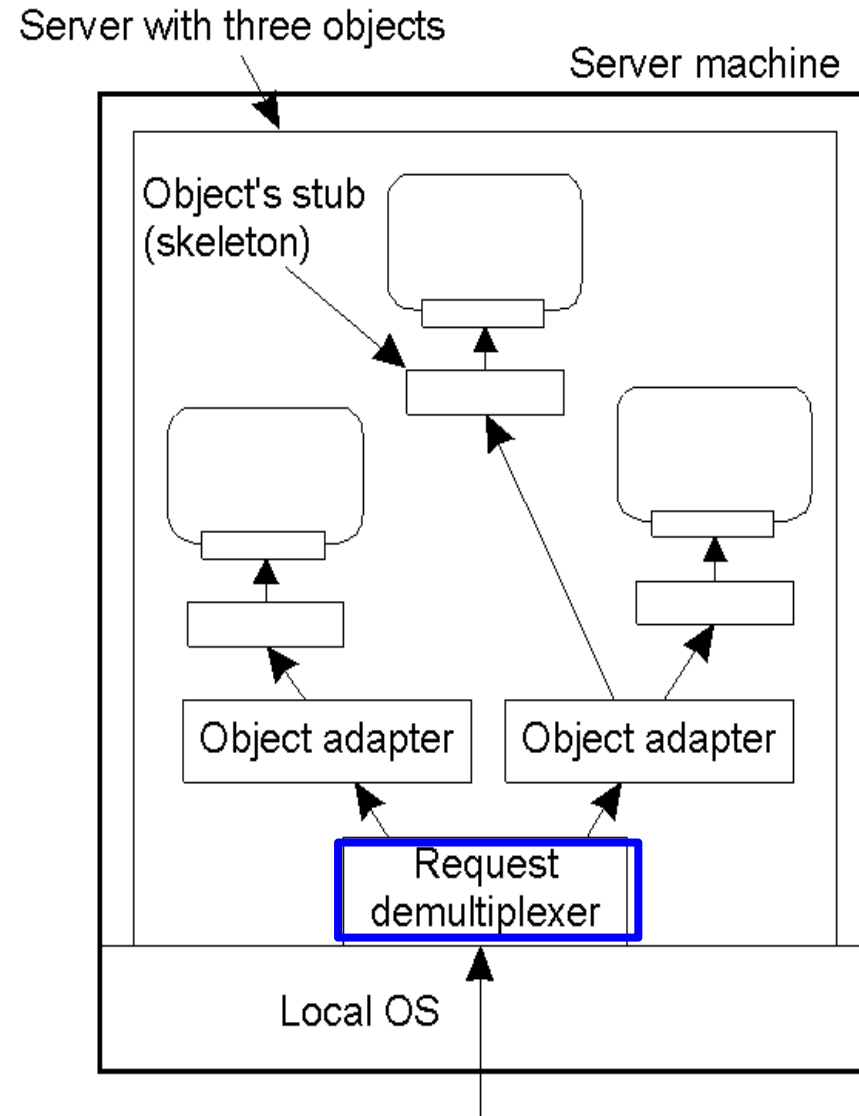
Finite State Machine



How to achieve more parallelism?

Object Adapter

- Organization of an object server supporting different activation policies via object adapters
- The **stub** at the callee site must be an active element, i.e. either a process or a task





Object Adapter (2)

```
/* Definitions needed by caller of adapter and adapter */
#define TRUE
#define MAX_DATA 65536

/* Definition of general message format */
struct message {
    long source                /* senders identity */
    long object_id;           /* identifier for the requested object */
    long method_id;          /* identifier for the requested method */
    unsigned size;           /* total bytes in list of parameters */
    char **data;             /* parameters as sequence of bytes */
};

/* General definition of operation to be called at skeleton of object */
typedef void (*METHOD_CALL)(unsigned, char* unsigned*, char**);

long register_object (METHOD_CALL call);        /* register an object */
void unrigester_object (long object)id);      /* unrigester an object */
void invoke_adapter (message *request);       /* call the adapter */
```

- The *header.h* file used by the adapter and any program that calls an adapter.



Object Adapter (3)

```
typedef struct thread THREAD;                /* hidden definition of a thread */  
thread *CREATE_THREAD (void (*body)(long tid), long thread_id);  
/* Create a thread by giving a pointer to a function that defines the actual */  
/* behavior of the thread, along with a thread identifier */  
  
void get_msg (unsigned *size, char **data);  
void put_msg(THREAD *receiver, unsigned size, char **data);  
/* Calling get_msg blocks the thread until of a message has been put into its */  
/* associated buffer. Putting a message in a thread's buffer is a nonblocking */  
/* operation. */
```

- The *thread.h* file used by the adapter for using threads.



Object Adapter (4)

- The main part of an adapter that implements a thread-per-object policy.

```
#include <header.h>
#include <thread.h>
#define MAX_OBJECTS    100
#define NULL           0
#define ANY            -1

METHOD_CALL invoke[MAX_OBJECTS]; /* array of pointers to stubs */
THREAD *root; /* demultiplexer thread */
THREAD *thread[MAX_OBJECTS]; /* one thread per object */

void thread_per_object(long object_id) {
    message *req, *res; /* request/response message */
    unsigned size; /* size of messages */
    char **results; /* array with all results */

    while(TRUE) {
        get_msg(&size, (char*) &req); /* block for invocation request */

        /* Pass request to the appropriate stub. The stub is assumed to
        /* allocate memory for storing the results.
        (invoke[object_id])(req->size, req->data, &size, results);

        res = malloc(sizeof(message)+size); /* create response message */
        res->object_id = object_id; /* identify object */
        res->method_id = req->method_id; /* identify method */
        res->size = size; /* set size of invocation results */
        memcpy(res->data, results, size); /* copy results into response */
        put_msg(root, sizeof(res), res); /* append response to buffer */
        free(req); /* free memory of request */
        free(*results); /* free memory of results */
    }
}

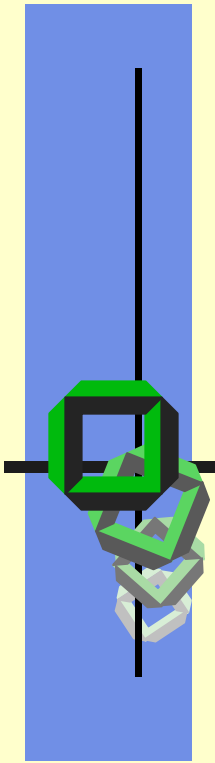
void invoke_adapter(long oid, message *request) {
    put_msg(thread[oid], sizeof(request), request);
}
```

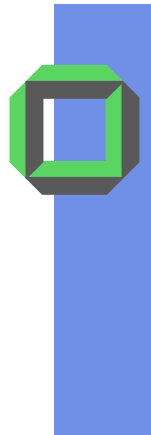
Client Templates

Teams

Pipelines

Master/Slave





Teams

- **Replicated** team members
 - Each member does the same

- **Specialized** team members
 - Some members do special things

- **Pipeline**
 - Each member does something special

- **Master/Slave**
 - Master manages and slaves do the work



Teams as Multithreaded Clients

- In WANs \exists high communication latency \Rightarrow
 - Use multiple communication threads to different partners instead of one communication facility
- Example: [Web browser](#)
 - Displays already text while other parts (often images, photos etc.) is not yet available
 - Parallel fetching of images & embedded links by special threads for CGI, ...
 - Several concurrent connections to the same server
 - ... or parallel transfers from replicated servers



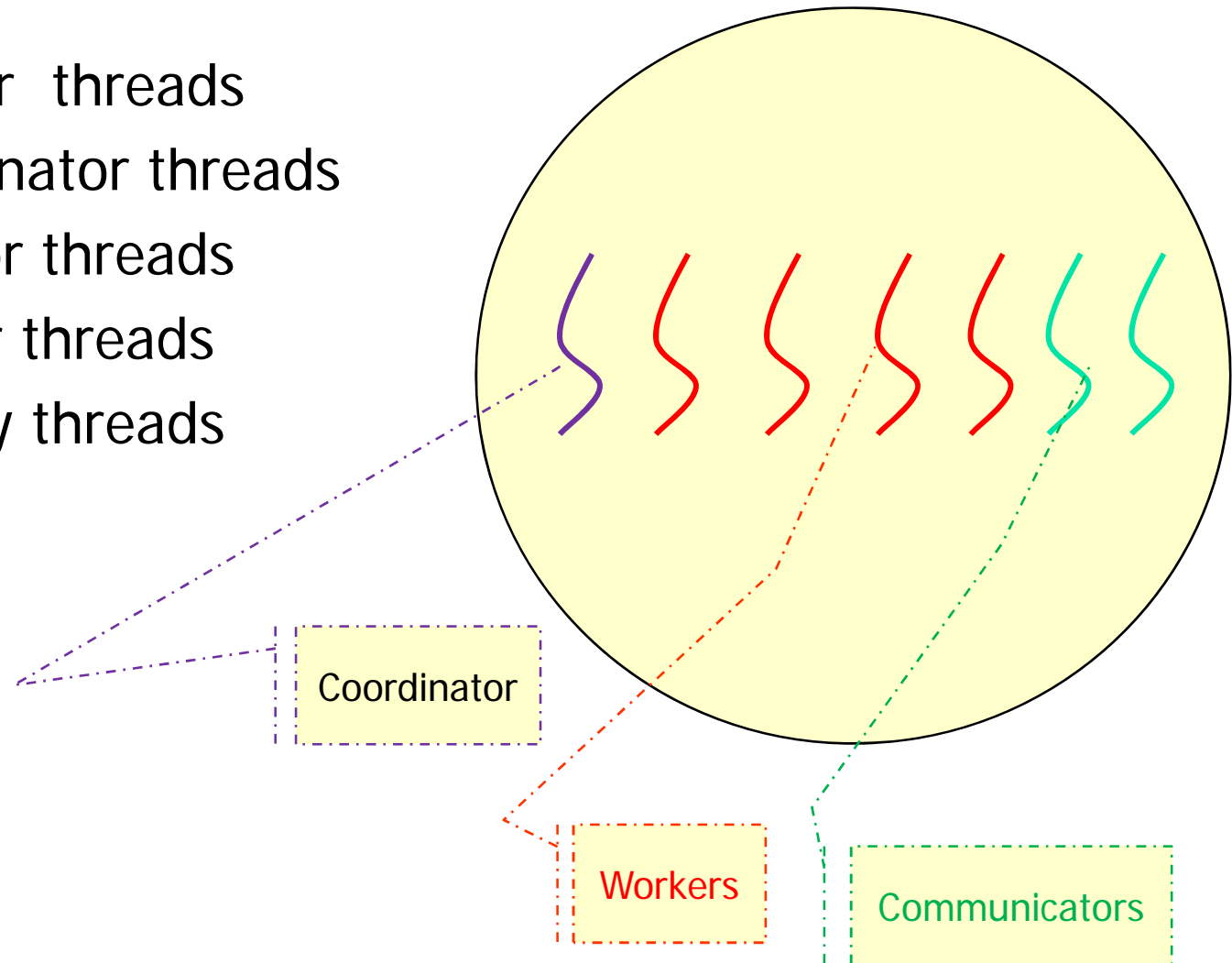
Replicated Team Threads

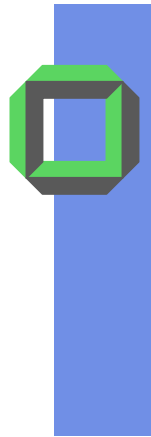
- Chess program with $n > 1$ threads analyzing the best move using a chess DB or even different chess DBs
- Simulation of a population of $n \gg 1$ subjects
- Numeric solutions of difference equations ...
- Periodic state reports (**still alive messages**), one for each connected partner node
- ...



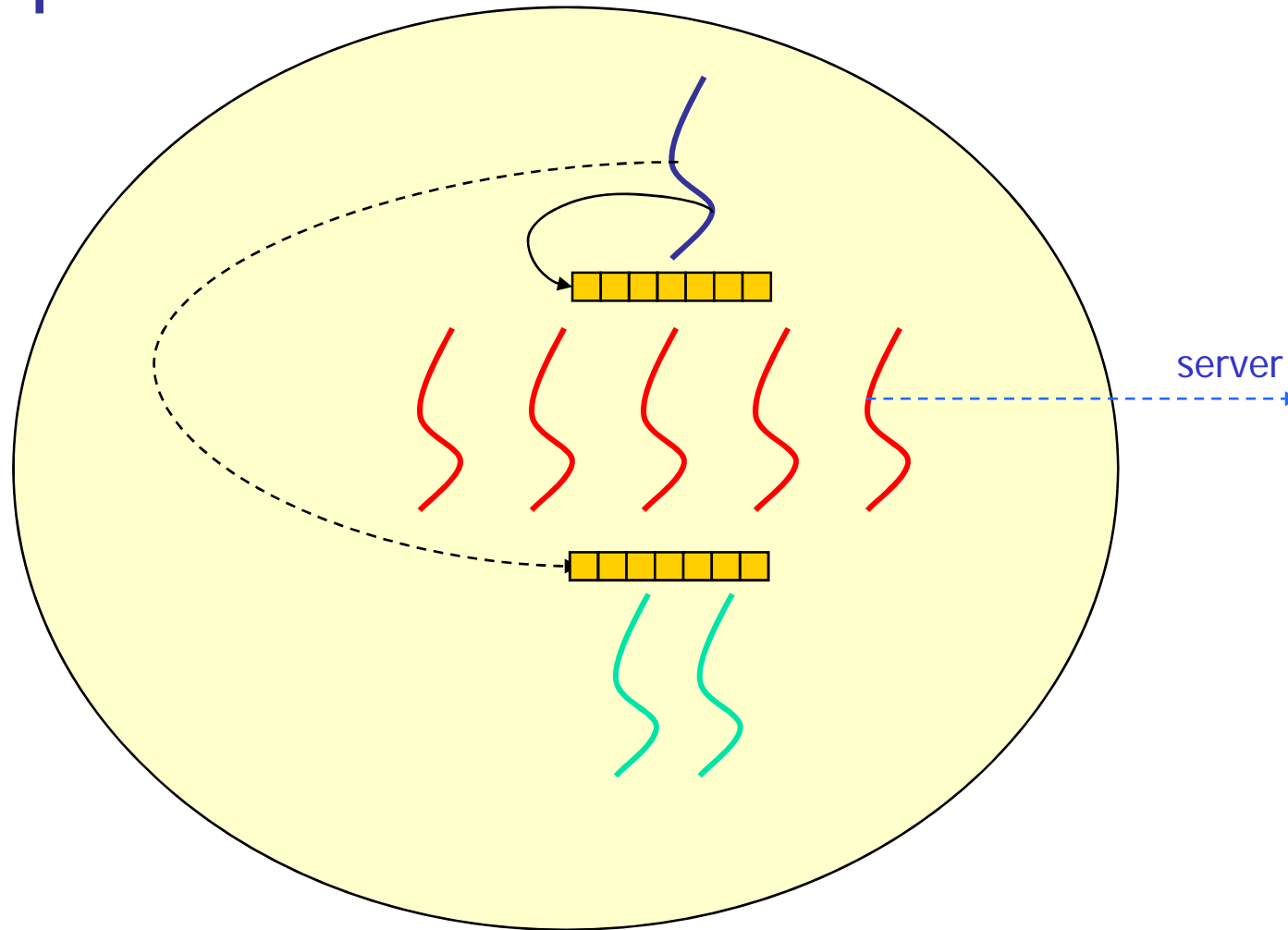
Specialized Teams

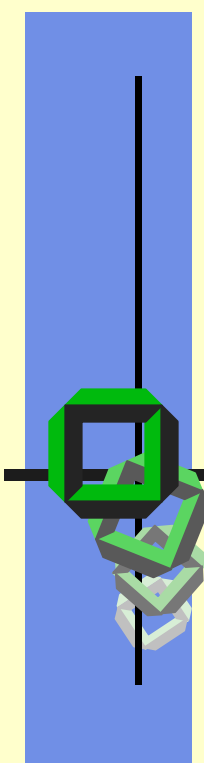
- Worker threads
- Coordinator threads
- Monitor threads
- Sensor threads
- Display threads
- ...





Pipelined Threads





Examples of Clients

X Window System

Thin-Client Network Computing

Compound Documents

Client-Side Software for Distribution

Transparency



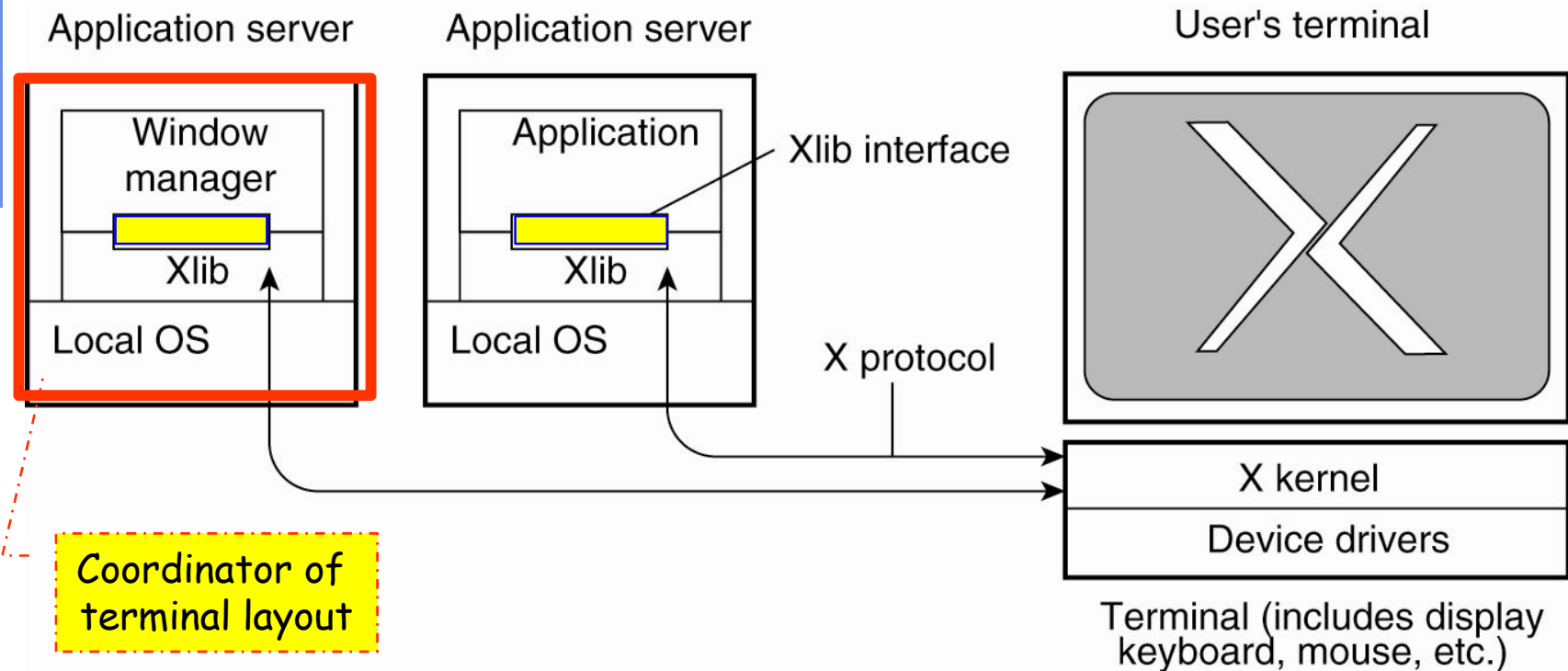
Clients for User Interfaces

Typical client tasks

- Offer a comfortable **user** interface, e.g.
 - X-windows
 - compound documents
- Client-side software for distribution transparency



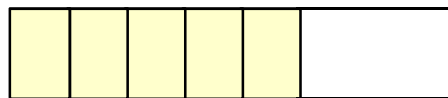
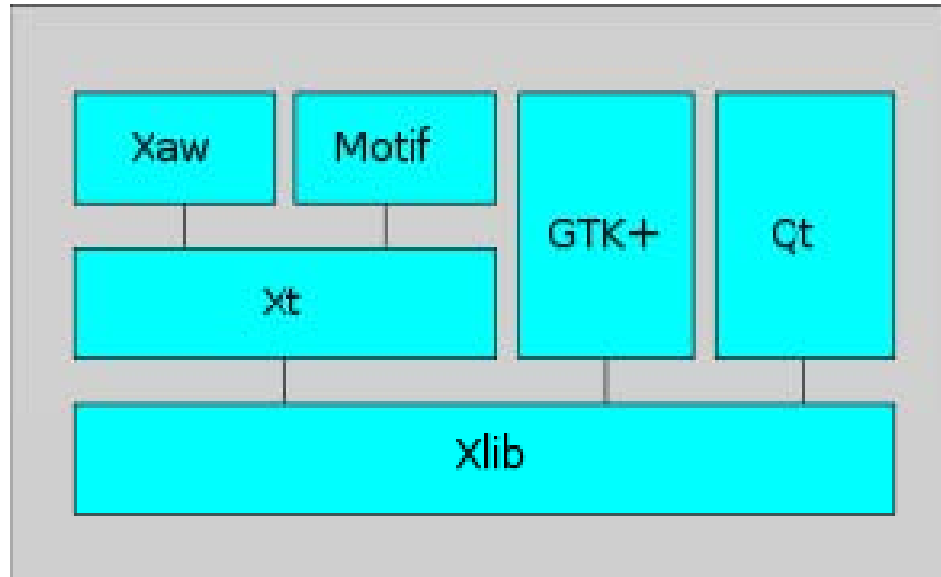
The X Window System



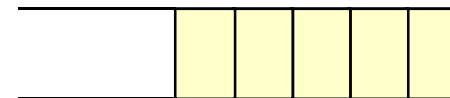
- Basic organization of the X-Window System
- X-kernel acts as a server receiving requests from clients, i.e. the application server
- X-kernel is implemented on the client's machine



XLIB



Event queue from X-server



Request queue to X-server



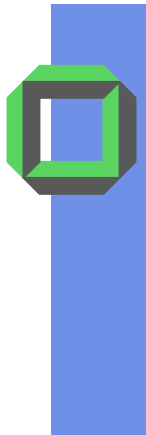
Typical XLIB-Functions

- operations on the connection, e.g.
 - `XOpenDisplay`, `XCloseDisplay`
- requests to the server,
 - including requests for operations, e.g. `XCreateWindow`, `XCreateGC`, ... or
 - requests for information, e.g. `XGetWindowProperty`
- operations that are local to the client,
 - operations on the event queue, e.g. `XNextEvent`, `XPeekEvent`
 - other operations on local data, e.g. `XLookupKeysym`, `XParseGeometry`, `XSetRegion`, `XCreateImage`, `XSaveContext`,



Example

```
// Xlib application drawing a little back box in a window
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
    Display *d;
    Window w;
    XEvent e;
    char *msg = "Hello, World!";
    int s;
        // open connection with the server
    d = XOpenDisplay(NULL);
    if (d == NULL) {
        fprintf(stderr, "Cannot open display\n");
        exit(1); }
}
```



```
s = DefaultScreen(d);
                                //create window
w = XCreateSimpleWindow(d, RootWindow(d, s), 10, 10, 100, 100, 1,
                        BlackPixel(d, s), WhitePixel(d, s));
                                // select kind of events we are interested in
XSelectInput(d, w, ExposureMask | KeyPressMask);
                                // map (show) the window
XMapWindow(d, w);

                                // event loop
while (1) {
    XNextEvent(d, &e);
                                //draw or redraw the window
    if (e.type == Expose) {
        XFillRectangle(d, w, DefaultGC(d, s), 20, 20, 10, 10);
        XDrawString(d, w, DefaultGC(d, s), 50, 50, msg, strlen(msg));
    }
                                // exit on key press
    if (e.type == KeyPress)
        break;
}

                                // close connection to server
XCLOSEDisplay(d);
return 0;
}
```

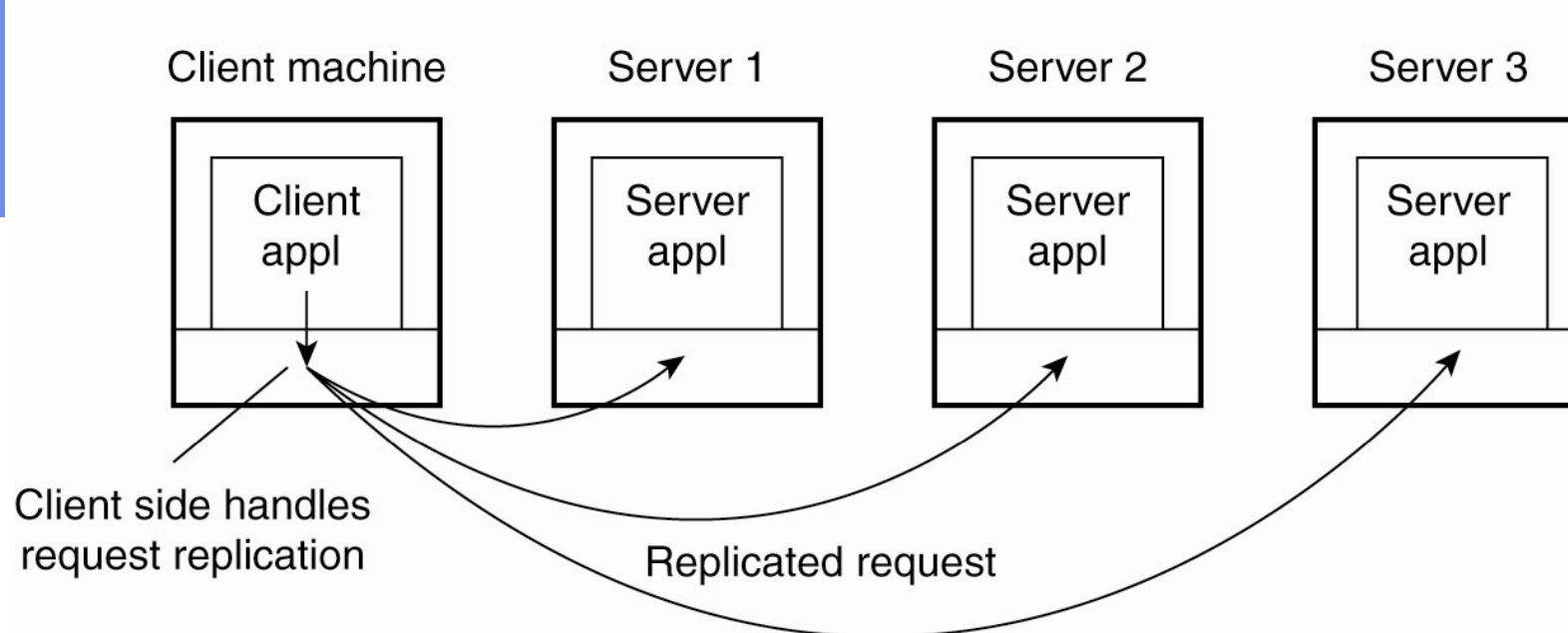


Client-Side Software for Distribution Transparency

- Besides pure user interfacing at the client side, some specific application code might be appropriate in client-server systems at the client side
 - Client software for ATMs, bar code readers ...
 - Stub handle access transparency, e.g. hiding heterogeneity of the node architectures
 - Client software can also handle location, migration transparency, i.e. it is informed when a server has changed its location or when it has shut-down
- In at least some of these cases, threads might be the right solution



Client and Distribution Transparency

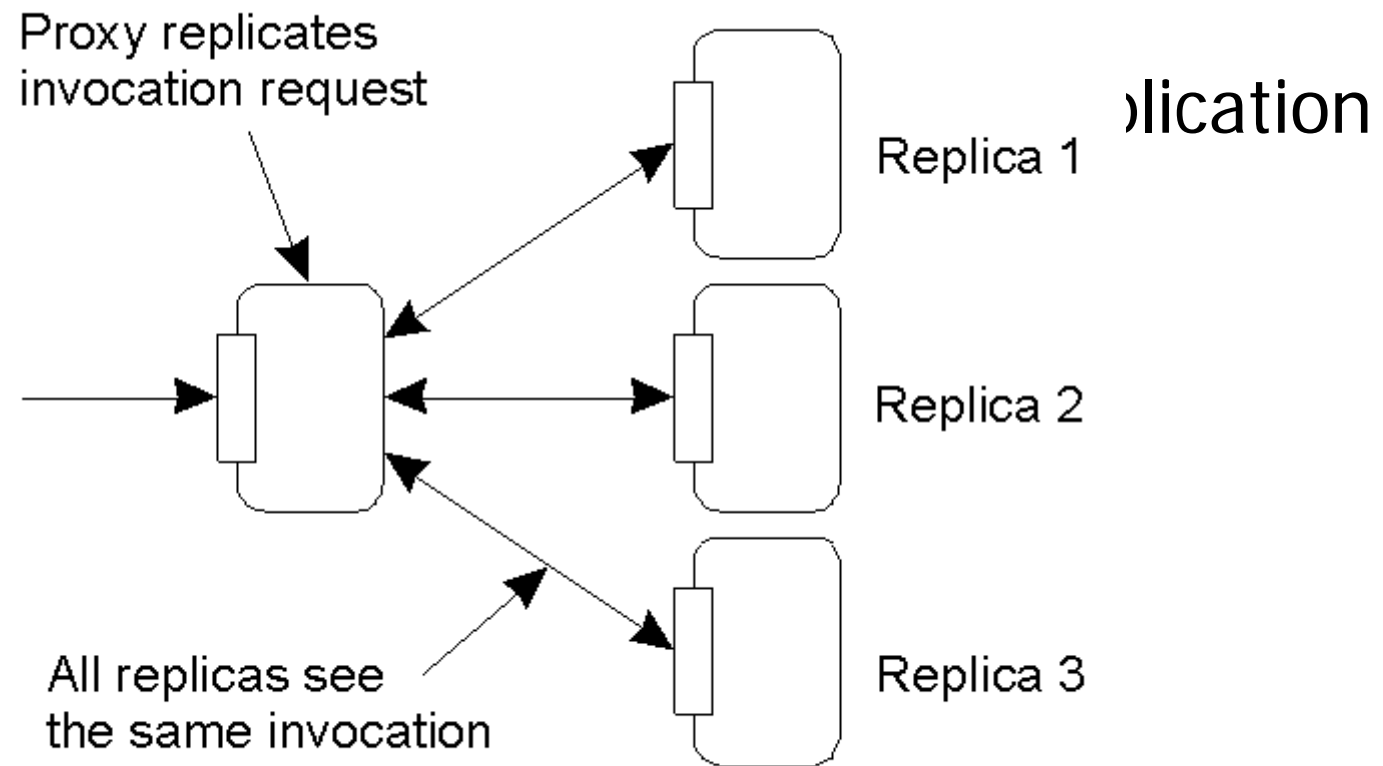


- Multicast each request to all servers and take the result according to some **policy**, e.g.
 - take result of the server that is replying first
 - take result of the majority
 - take into account misbehaving or missing servers
 - ...



Client-Side Software for Distribution Transparency

- A of SO

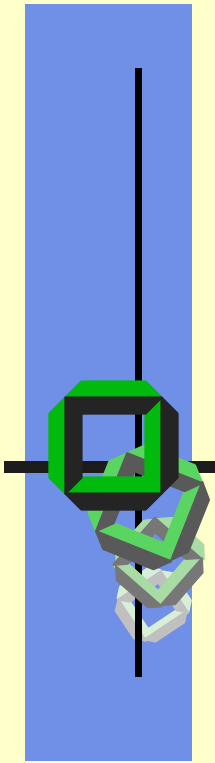


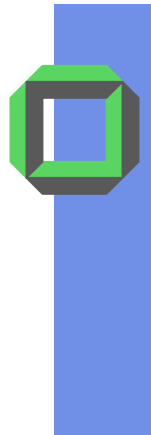


Clients and Failure Transparency

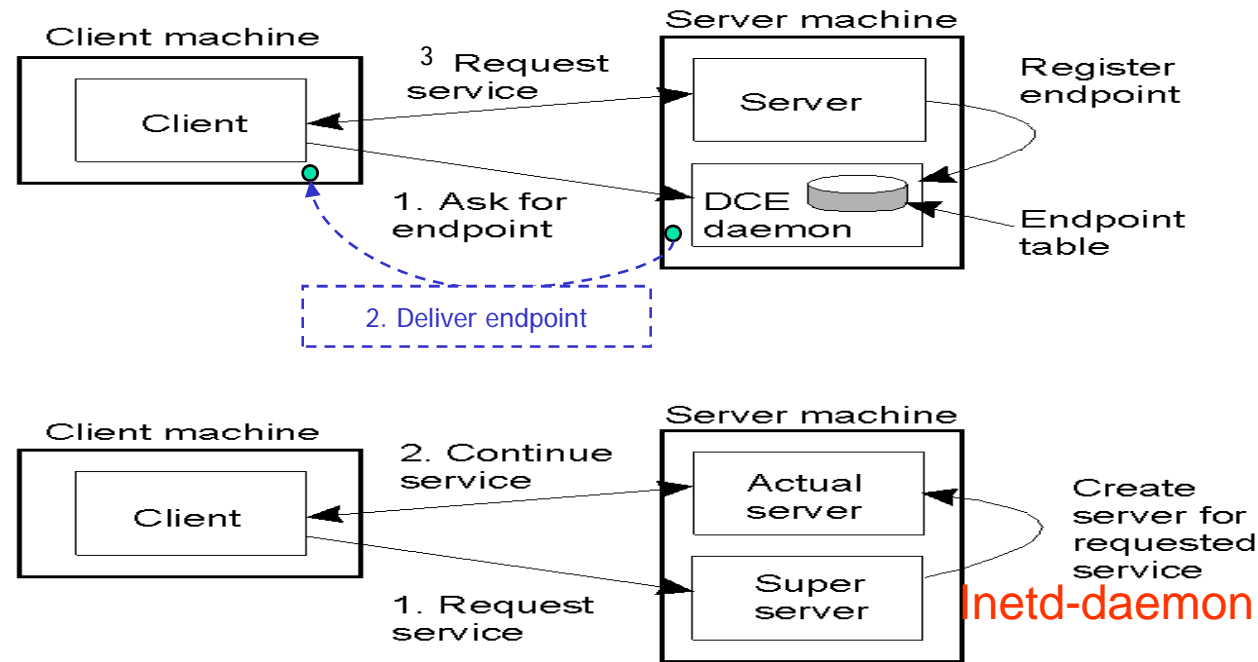
- Masking communication failures with some server is typically done by client's middleware (MW), e.g.
 - Client's MW could repeatedly try to connect to a server or
 - perhaps try to contact another server after several attempts
 - Client's MW could also use cached information of a previous session
 - Web browsers can do so if web server is currently not available

Server Templates





Servers: General Design Issues



(b)

- a) Client-to-server binding via a daemon as in DCE who knows all co-located end points of the current servers at that node
- b) Client-to-server binding via a superserver as in UNIX, i.e. the super server listens to **all incoming requests** and forks a special server responsible for that request



Server Templates

- Each server installs a specific location (end point) where it can wait for incoming requests, e.g. a port, a channel, ...
- 2 basic templates of servers:
 - Serial server process
 - Server waits for the request, handles service, and finally returns result to a client before accepting next request
 - Concurrent server (e.g. a multithreaded server)
 - Waiting done by dispatcher thread
 - Each service by worker threads



Server Endpoints

- Client sends request to server, e.g. to server **port**
- How do clients know these endpoints?
 1. Assign **specific endpoints** for well-known services
 - FTP port 20/21
 - Telnet port 23
 - Simple Mail Transfer port 25
 - Login host port 49
 - HTTP port
 2. **Name services** will support clients to find machine addresses with corresponding services



Analysis of a Serial Server Process

Major Drawback

- Single point of failure
- Low performance, no internal concurrency ⇒
 - Communication latency
 - Increased turn around time of client's requested service
- If multiple clients, you need a request buffer
 - You need a request buffer anyway
- Restricted server policy
 - Often only FCFS policy
 - Others might be better
 - *Any chance to sail around an inherent FCFS?*



Adaptive Server Processes

- Assume a monitor instance controlling the load of a server process
 - In case of an overload install an additional server process
 - In case of underload detach a server process for a while
 - *How to notice over- or underload?*
 - *When to measure what queue?*
 - Internal request queue often can not be inspected
 - Request queue in front of server might be not complete



Analysis of a Multiplexed¹ Server

- Advantages
 - A bit of concurrency, depending on whether you can avoid blocking within the server
- Disadvantages
 - Complicated switching between different select phases

¹Finite state machine



Analysis of a Multiprocessing Server

■ Advantages

- Concurrency (usage of multiprocessors)
- Usage of blocking time of other processes
- Better average turnaround time
- No extra switching within the server, uses the standard kernel scheduler

■ Disadvantages

- Forking is expensive
- Need for shared-memory support



Analysis of a Multithreaded Server

- Advantages
 - If KLT model \Rightarrow full server concurrency
 - In case of blocking I/Os, only KLT or hybrid model useful
 - No need for shared-memory support
- Disadvantages
 - Need coordination of threads iff accessing common data
 - KLT cloning still expensive¹
 - **Upper limit** of number of worker threads?²
 - Each additional thread needs memory (TCB, STACK)

¹ using a predefined pool of workers might help

² pool size might be a tuning parameter

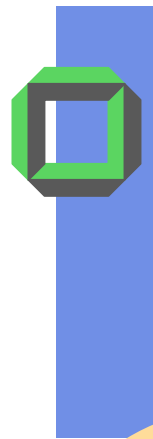


Multithreaded Server Models

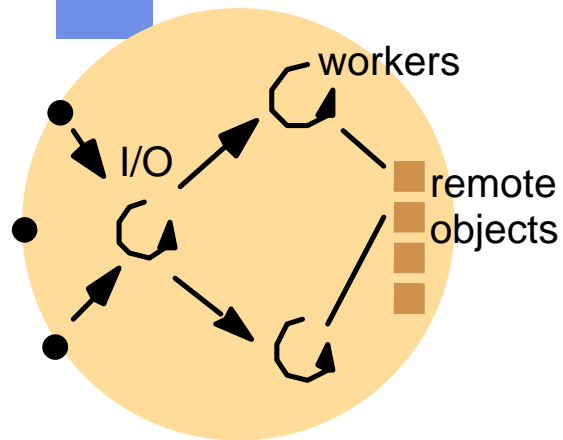
1. Thread per request
 - thread creation/switching overhead high
 - Must limit # threads in case of overload

2. Thread per connection
 - somewhat less overhead
 - Must limit the number of threads in case of overload

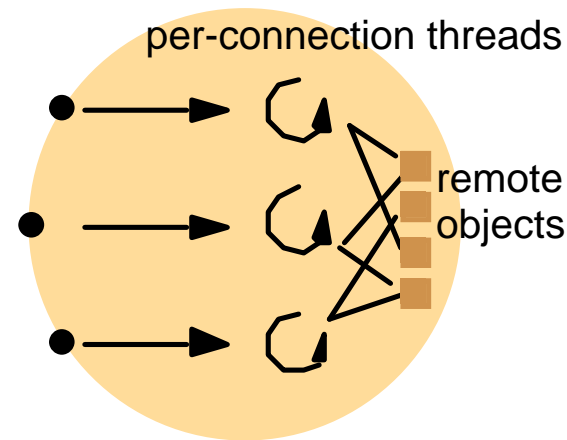
3. Thread per object
 - Simplifies synchronization
 - Sequential execution per object



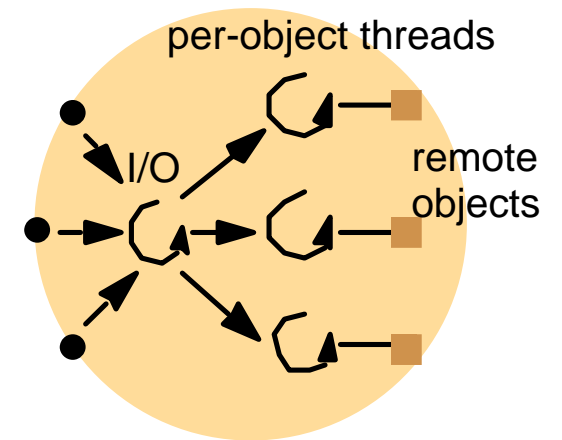
Alternative Object Server Threading



a. Thread-per-request



b. Thread-per-connection



c. Thread-per-object



Interruptible Server

Scenario:

User just has started to upload a big file to an ftp-server. Soon after she/he notices: "Oops", it was the wrong file. *How to stop this superfluous file transmission?*

1. Approach: Internet user exits the client application \Rightarrow automatically breaks connection to ftp-server, then restarts it. The server will tear down old connection, thinking that client has had a crash.
2. Solution: client sends an **out-of-band message**
 1. At a separate control endpoint & with a separate urgency thread
 2. At normal port, but message is classified as **urgent** and received and interpreted by dispatcher thread who can abort the upload



Stateless versus Stateful Servers

- **Stateless Server** (e.g. http, NFS)
 - No bookkeeping of the states of its clients
 - It may change its state without informing the clients \Rightarrow recovery is easy
 - However, even a stateless server can maintain some clients' states in a client log in order to be able to replicate itself **appropriately**
 - If this log is lost, well the server still can work
- **Semi stateless server** (soft state model)
 - Client state is maintained but only for a limited time



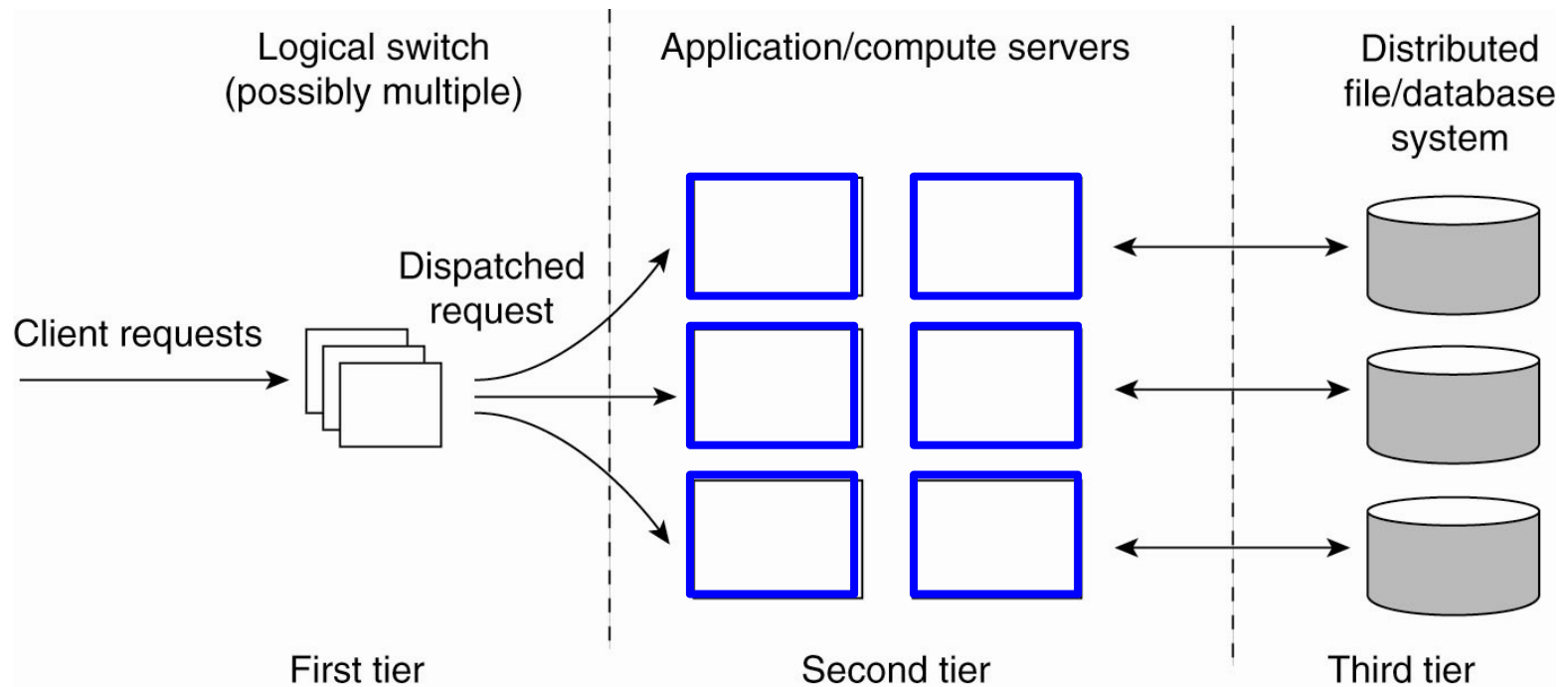
Stateless versus Stateful Servers

- Stateful server

- keeps information on clients' behavior and state, e.g. a file-server that allows a client to keep a local copy of a file, even for updating that file
 - May improve control of which client is allowed to update the file \Rightarrow knowing where the actual version of the file is
- Recovery is more complex
- Client's access history etc. kept in a cookies sent by the web server to the web browser in order to support future accesses



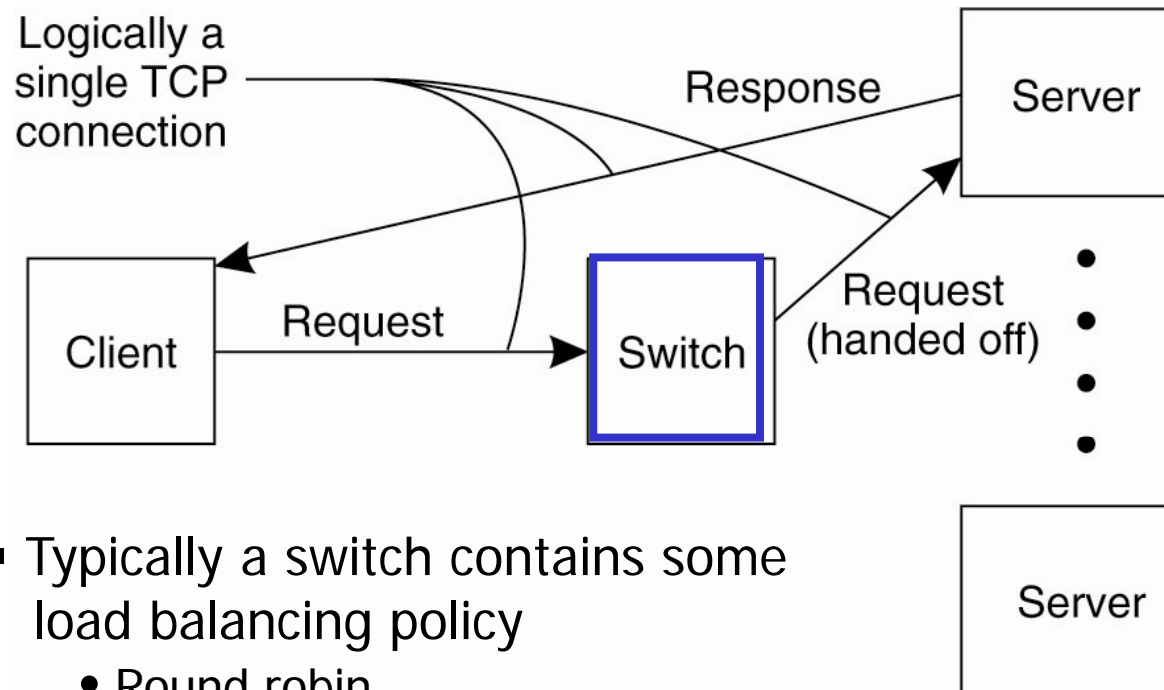
Server Clusters (1)



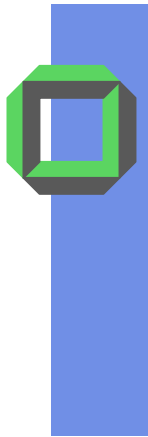
- General organization of a 3-tiered server cluster in a LAN
- First tier is a switch forwarding clients' requests to the appropriate application server



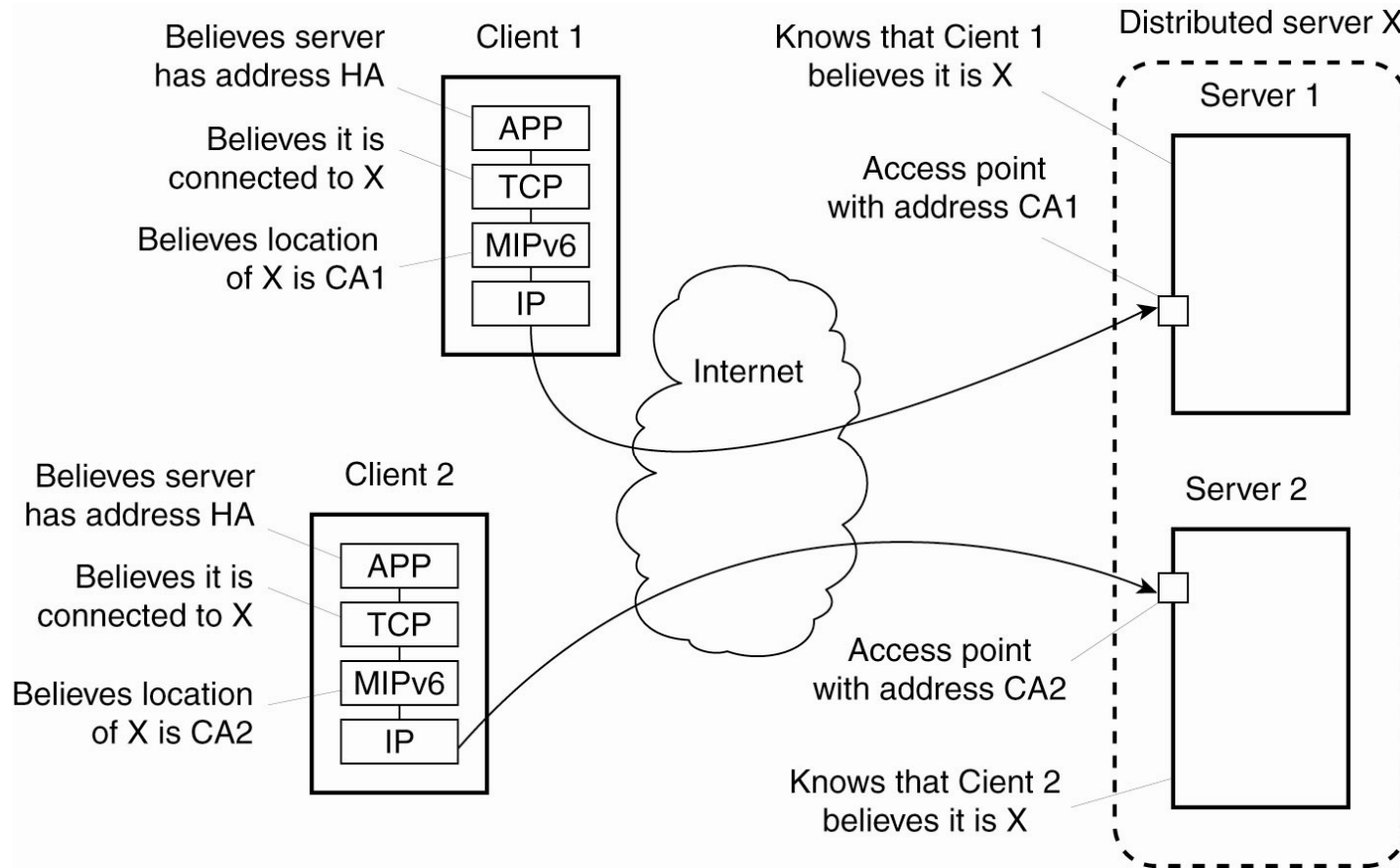
Server Clusters (2)



- Typically a switch contains some load balancing policy
 - Round robin
 - Content aware scheduling
- Access transparency at the client should be provided. A client is not interested in the internal structure of the cluster
- A server cluster might have multiple entry-points, i.e. switches
- A client opens a TCP connection and closes it after the reply from one of the servers



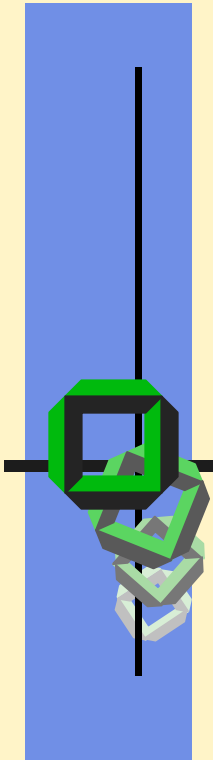
Distributed Servers



- Route optimization in a distributed server

Client-/Server Templates

Architectures



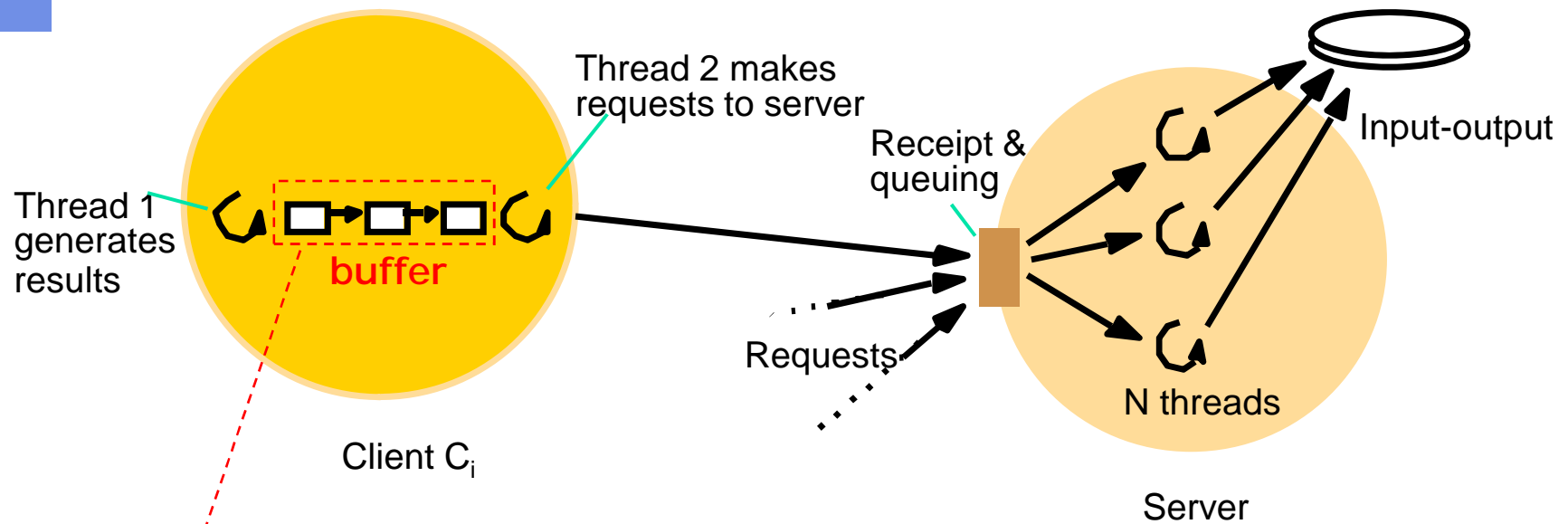


Review: Client-Server Concept

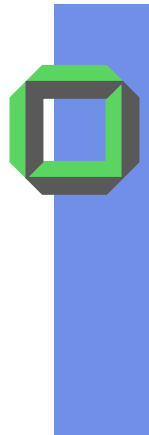
- Server requested by many clients
- RPC protocol typically used to issue requests
- Server can
 - manage special data
 - run on an especially fast platform
 - have an especially large disk
- Client handles “front-end” processing and interaction with the human user



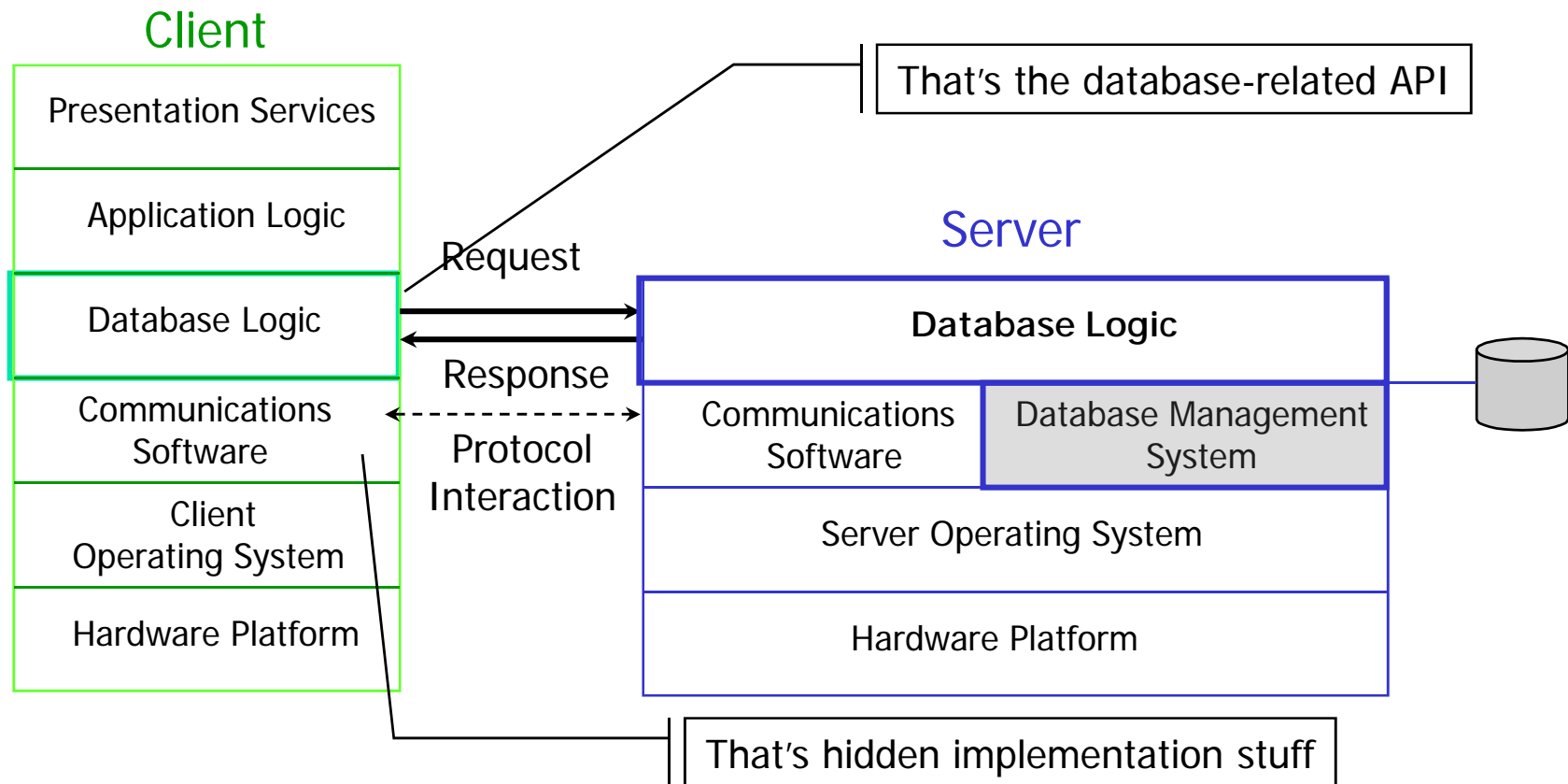
Pipelined Client & Master/Slave Server



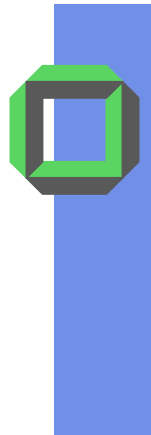
Why do we use a buffer at client's node?



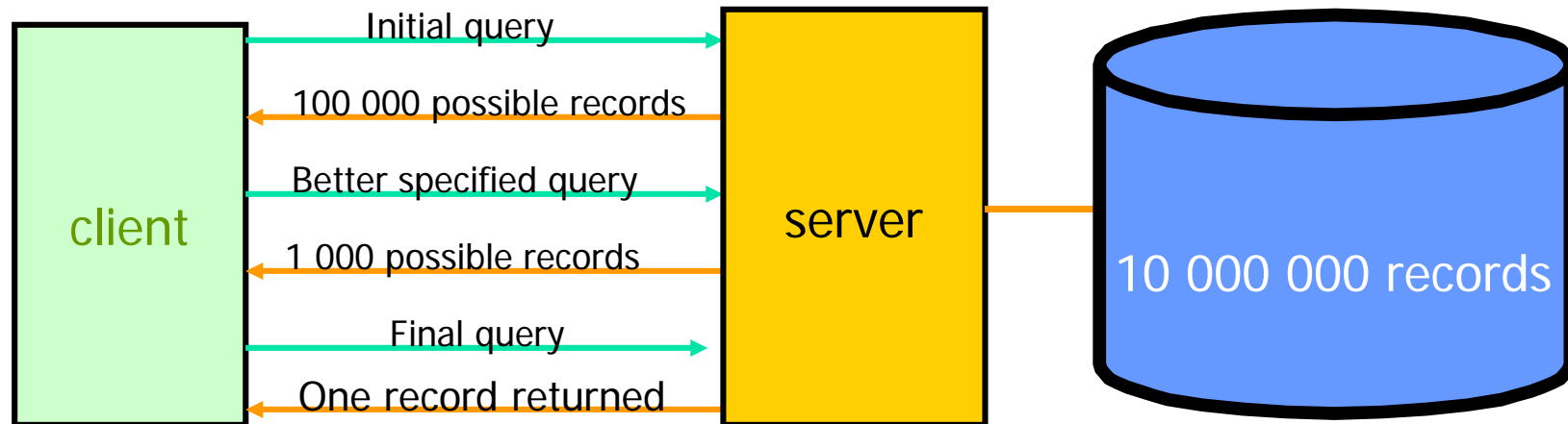
Client/Server Templates



Remark: Often, only the **DB-related functions** are implemented on the server.



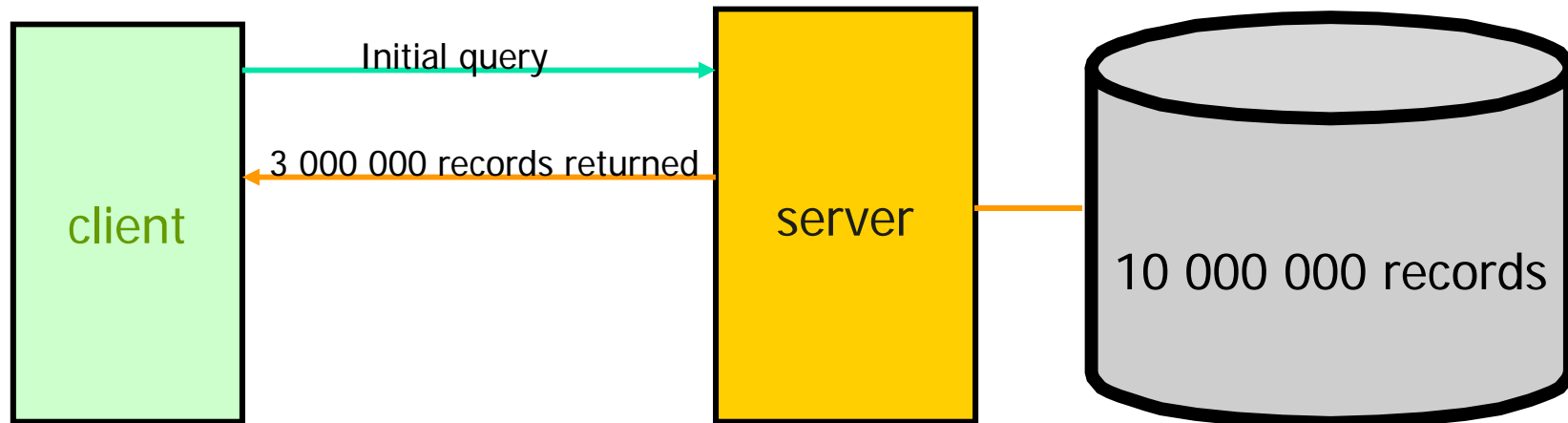
Appropriate Client/Server Pattern



1. Massive job of sorting and searching within the database, requiring resources being too expensive for each client
2. Too much traffic on the net to move the entire data base to a client in order to do a local search and sort.



Misused Client/Server Pattern



The client wishes to find the mean value across many records of the database.

Implementing the above application function within the server would solve the problem.



Examples of Servers

- Network file server
- Database server
- Network information server
- Domain name service
- Microsoft Exchange server
- Kerberos authentication server



Business Examples

- Risk manager for a bank: tracks exposures in various currencies or risk in investments
- Theoretical price for securities or bonds: traders use this to decide what to buy and what to sell
- Server for an ATM: decides if your withdrawal will be authorized



Bond Pricing Example

- Server receives market trading information, currency data, interest rates data
- Has a database of all the bonds on the market
- Client expresses interest in a given bond, or in finding a bond with certain properties
- Server calculates what that bond (or what each bond) should cost given current conditions



Why use a Client-Server Approach?

- Pricing parameters are “expensive” (in terms of computing resources) to obtain: must monitor many data sources and precompute many time-value of money projections for each bond
- Computing demands may be extreme: demands a very high performance machine
- Database of bonds is huge: large storage, more precomputation



On Client Side

- Need a lot of CPU and graphics power to display the data and interact with the user
- Dedicated computation provides snappy response time and powerful decision making aids
- Can “cache” or “save” results of old computations so that if user revisits them, won't need to reissue identical request to server



Summary of a Typical Split

- Server deals with
 - bulk data storage
 - high performance computation
 - collecting huge amounts of background data that may be useful to any of several clients
- Client deals with the “attractive” display, quick interaction times
- Use of caching to speed response time



Statefulness Issues

- **Client-Server system is stateless if:**

Client is independently responsible for its actions, server doesn't track set of clients or ensure that cached data stays up to date

- *Client-server system is stateful if:*

Server tracks its clients, takes actions to keep their cached states "current". Client can trust its cached data.



Best known Examples?

- The UNIX NFS file system is stateless.

Bill Joy: "Once they replaced my file server during the evening while my machine was idle. The next day I resumed work right where I had left off, and didn't even notice the change!"

- Database systems are usually stateful:

Client reads database of available seats on plane, information stays valid during transaction



Typical Issues in Design

- A client is often much simpler than a server: a client **can be single-threaded**, can wait for reply to RPC's
- Server is generally multithreaded, designed to achieve extremely high concurrency and throughput. Much harder to develop
- **Reliability issue**: if server goes down, all its clients may be "stuck". Usually addressed with some form of **replication** or **backup**.



Use of Caching

- In stateless architectures, the cache is responsibility of the client. Client decides to remember results of queries and reuse them.

Examples:

- Caching Web proxies
 - NFS client-side cache.
-
- In stateful architectures, cache is owned by server. Server uses “callbacks” to its clients to inform them if cached data changes, becomes invalid. Cache is “shared state” between them.



Butler Lampson's Advice

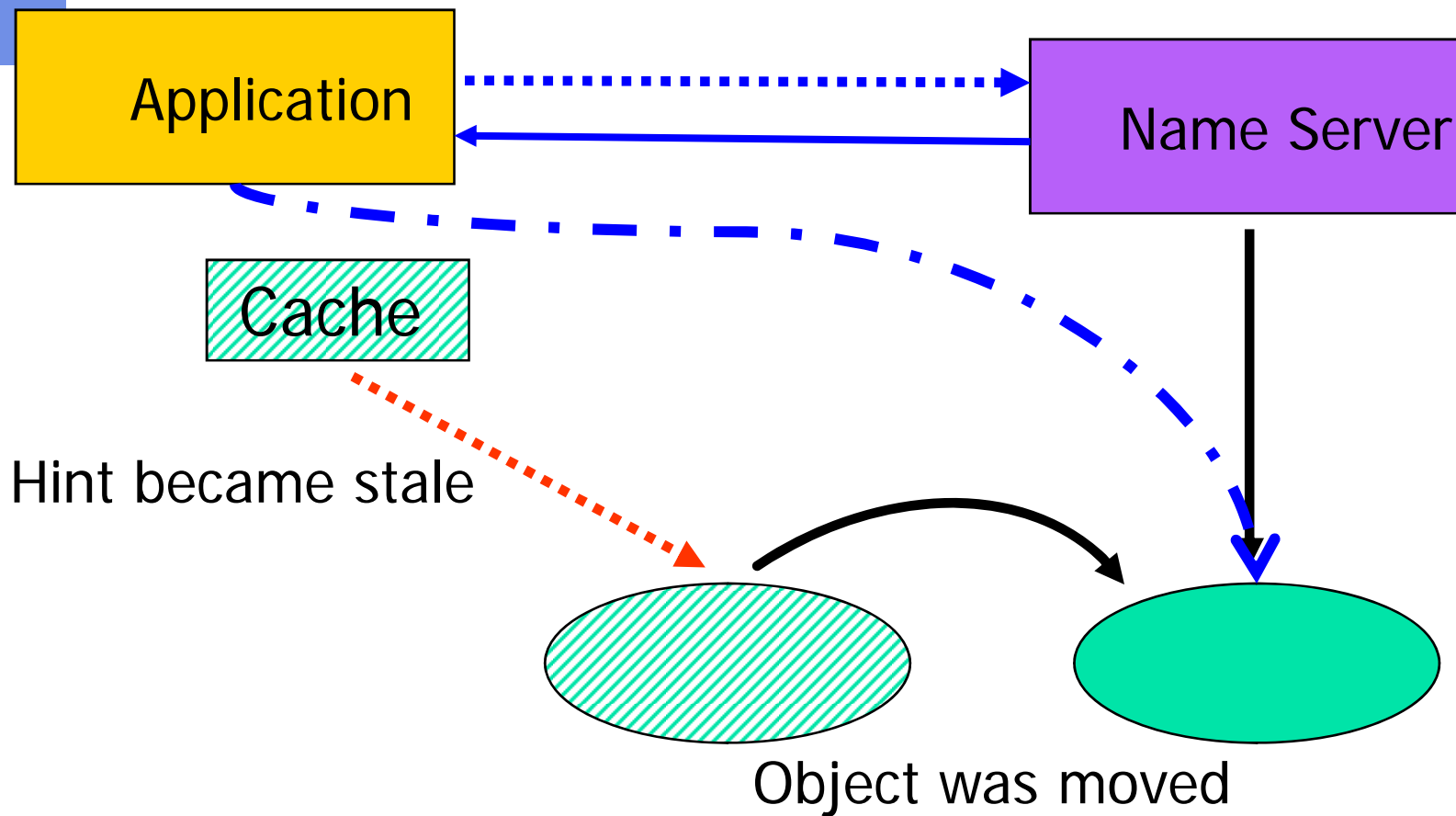
- Cache "hints"
 - Speed up system when hint is correct
 - A mechanism can detect when hint is wrong/stale and seeks for more up to date information
- If cached data is viewed as hints, relationship of client and server can be stateless

Example:

- information about location of a mailbox
- if hint is wrong, you just can run a more costly protocol



Butler Lampson's Advice





Example of Stateless Approach

NFS is stateless:

- clients obtain “vnodes” when opening files
- server hands out vnodes, but treats each file operation as a separate event

- NFS trusts:
 - vnode information
 - user’s claimed machine id
 - user’s claimed uid

- Client uses write-through caching policy

- Cache may be stale if someone writes the file after it is opened



Example of stateful approach

- Transactional software structure:
 - Data manager holds database
 - Transaction manager does begin op1 op2 ... opn commit
 - Transaction can also abort; abort is default on failure
- Transaction on database system:
 - Atomic: all or nothing effects
 - Concurrent: can run many transactions at same time
 - Independent: concurrent transactions don't interfere
 - Durable: once committed, results are persistent



Comments on Transactions

- Well matched to database applications
- Requires special programming style
- Typically, splits operations into read() and update() categories. Transactional architecture can distinguish these
- Idea is to run transactions concurrently, but to make it look as if they ran one by one in some sequential (serial) order



Why are Transactions Stateful?

- Client knows what updates it has done, what locks it holds. Database knows this too
- Client and database share the guarantees of the model. See consistent states
- Approach is free of the inconsistencies and potential errors observed in NFS



Tradeoffs...

- When we build a reliable application we need to start by asking what **reliability goals** we have in mind
 - Then we can try to map **them**
 - to a stateless client-server paradigm or
 - a transactional paradigm
 - Sometimes this won't work – some applications need more – but in such cases there are other ways to implement the needed functions
 - Each approach brings advantages and disadvantages and we are often forced to make tradeoffs



Examples of Tradeoffs?

- Stateless approaches are easier to build, but we know much less about the state in which the system may find itself after a crash
- Transactional approaches are well supported but really assume a database style of application (data lives “in” the database)
- Other mechanisms we’ll explore often force us to implement hand-crafted solutions in our applications and depart from the platform architectural standards (or go “beyond” them).
 - Companies hesitate when faced with such options because they are costly to support



Performance: Monster in the Closet

- Often, the hidden but **huge issue** is that we want **high performance**
 - After all, a slow system costs more to operate and may drive users crazy!
- The issue is that some techniques seem simple and seem to do the trick, but are as much as thousands of times slower than other alternatives
 - Forcing us to use those alternatives... And perhaps driving us away from what the platforms support



Market Failures

- In fact, many researchers/developers see reliability in the broad sense as victim of a market failure
 - This means that we know how to solve the problem, but
 - Major platform vendors don't believe the market will pay for the solutions
- Another chicken and egg issue:
 - with a solution, you could use it, and your use would contribute to a market.
 - But lacking a solution of course there is no market...



Why do Vendors see this?

- They cite failures in the 1980's
 - High reliability platforms were costly to build and tricky to use, and only customers with very stringent needs opted for them
 - So the broader market – the 80% solution – went with less flexible, limited solutions
 - Companies that build platforms, like HP and IBM and Microsoft, learned the lesson and decided to aim for the 80% sweet spot



Could this change?

- As network systems continue to expand and mature
 - They are getting much larger, hence we need to automate reliability and repair
 - And they are facing bigger load bursts, forcing us to think more about replication
 - And operators of big commercial sites are seeking to offer better guarantees of response time, availability, etc
- All of this is creating a new market pressure for richer reliability options