Telecooperation Lab
Prof. Dr. Max Mühlhäuser

TK1: Distributed Systems - Programming & Algorithms

Chapter 2: Distributed Programming

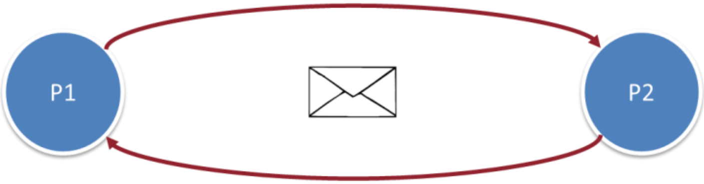
Section 1: Mainstream Paradigms

Lecturer: **Prof. Dr. Max Mühlhäuser, Dr. Immanuel Schweizer, Dr. Benedikt Schmidt**

Interprocess Communication (IPC)

Message passing:

- Sending messages between programs / processes
- Explicit communication



Socket:

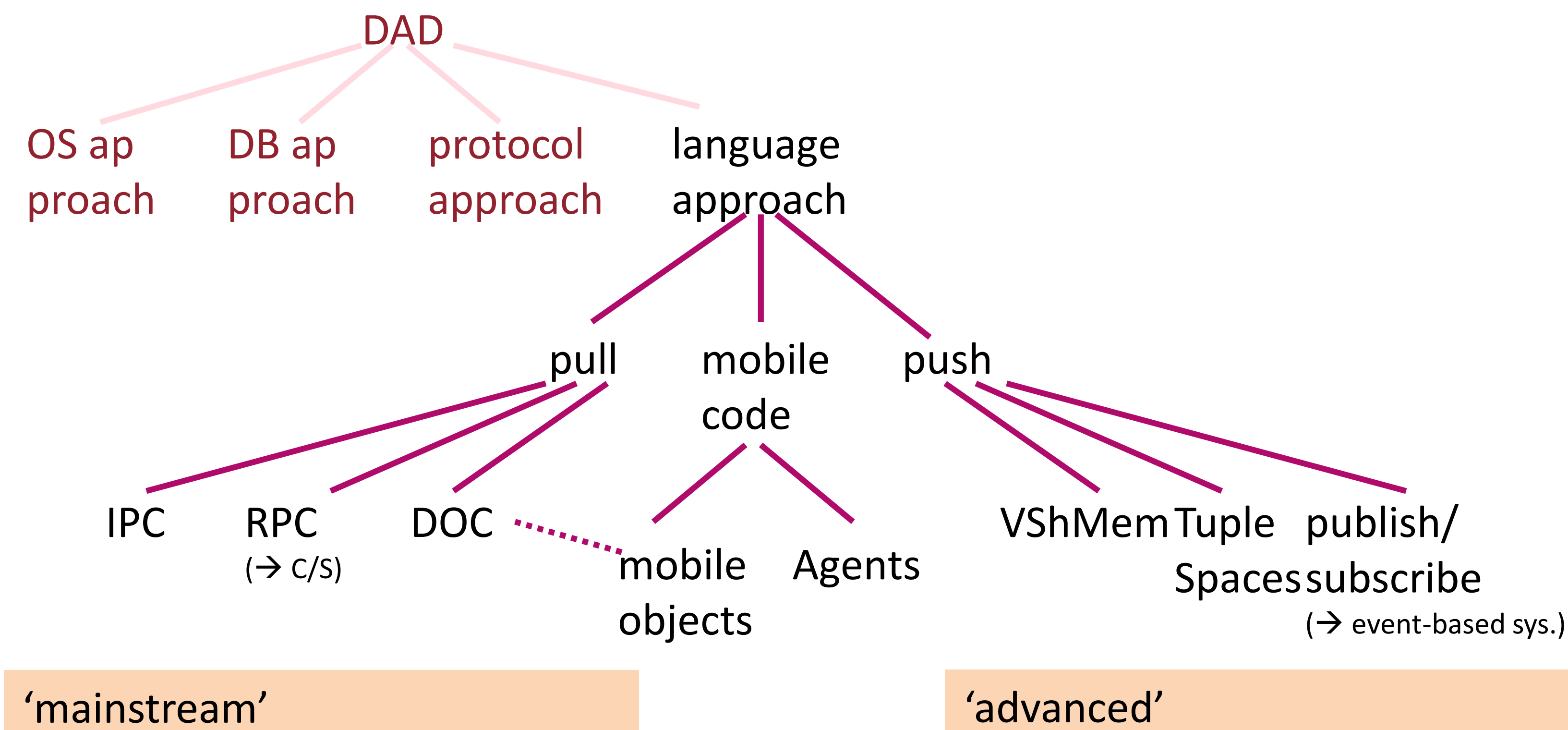
- „the API for the Internet“
- Internet Sockets
  - Connection-less (UDP) / Connection-oriented (TCP)
- Layer 4

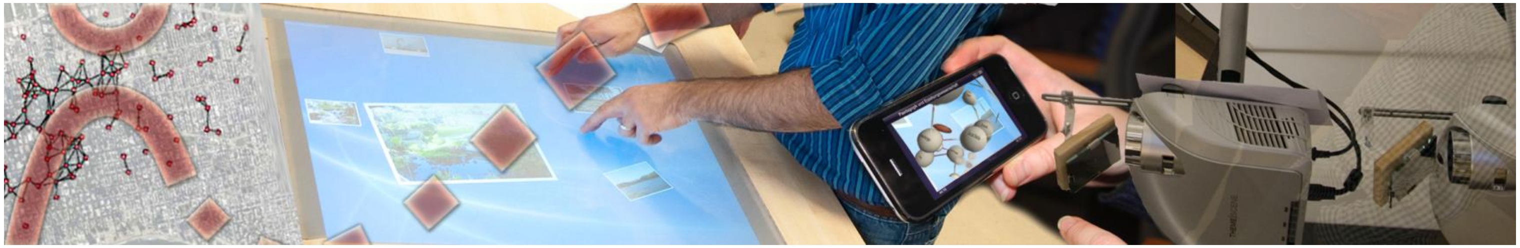
Message Queues:

- Persistent queues
- De-coupling
- Layer 7

Pragmatic Taxonomy

all in one, we get the following taxonomy for distributed application development (DAD):





2.1: MAINSTREAM PARADIGMS

(1) IPC: Interprocess Communication

(2) RPC: Remote Procedure Call

Remote Procedure Call (RPC)

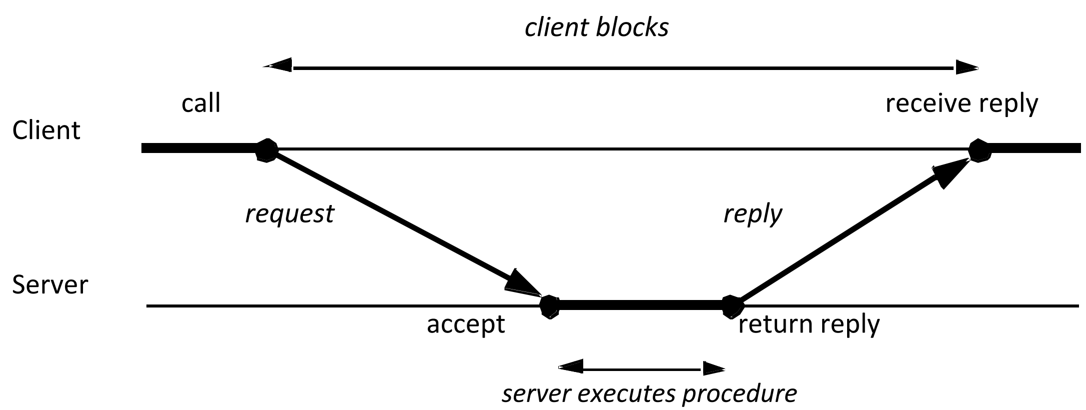
Fundamental **idea** behind RPC:

Processes can call procedures on other computers

- **Goal:** syntactic and semantic uniformity of local and remote calls, in terms of:
  - Call mechanism
  - “Expressive power“ of language
  - Error handling
- Goal cannot be fully achieved

Note: Regardless of problems, RPC is widely used in the computation world

RPC: Principle

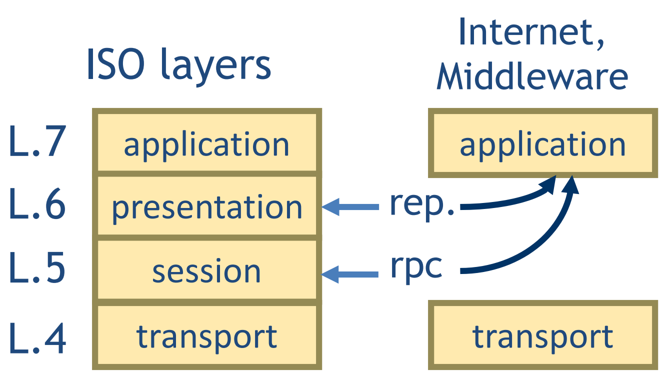


- Code in client:

```
Set out-parameters
Call X(out-parameters, result)
Use result
```

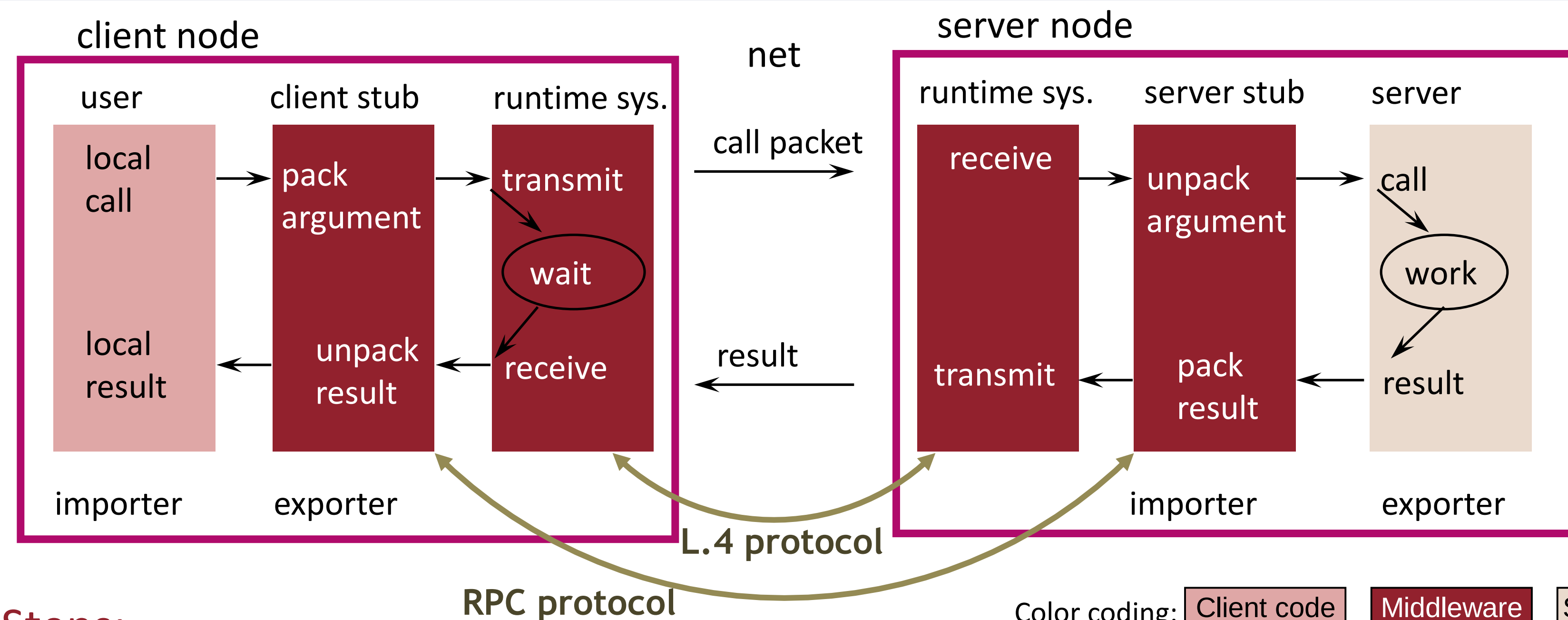
- Code on server:

```
Proc X(parameters)
Do stuff
Return (result)
```



rep.: common data representation
rpc: rpc protocol (*very* simple L.5)

RPC Control Flow

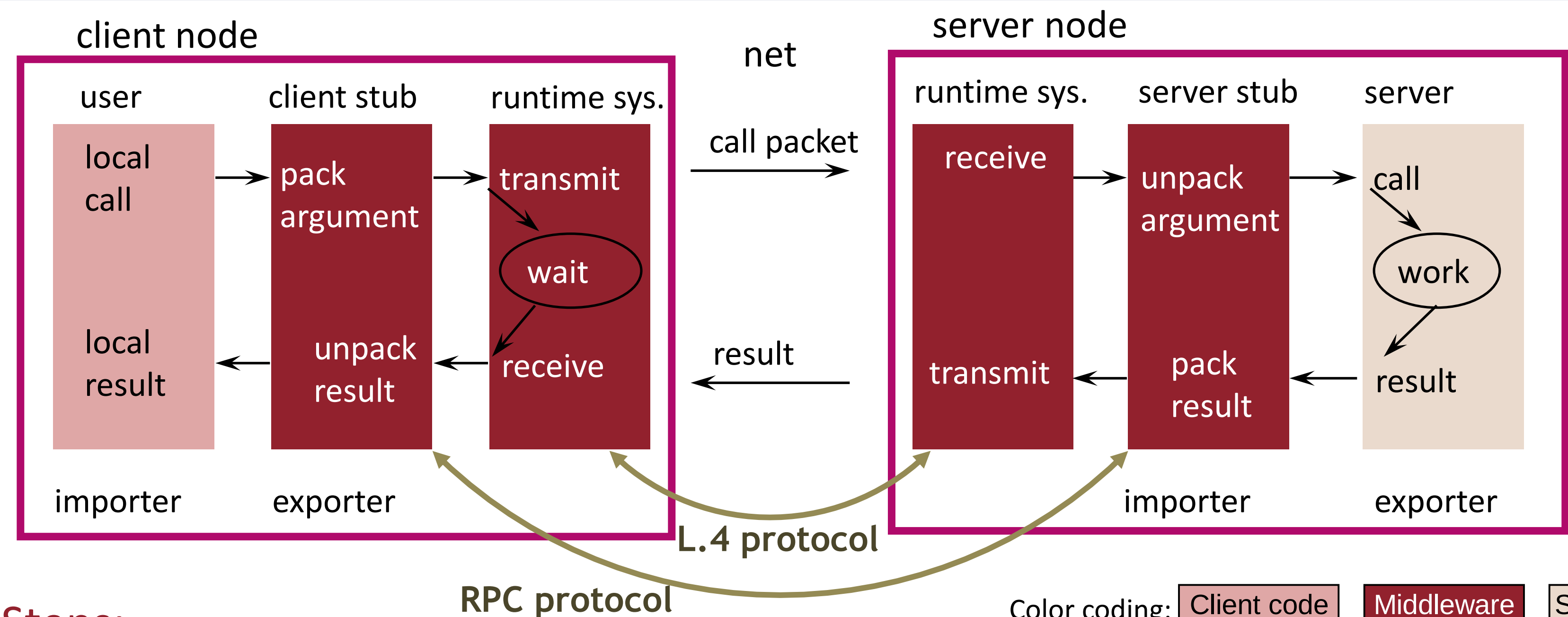


Color coding: Client code | Middleware | Server code

Steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.

RPC Control Flow



Color coding: Client code | Middleware | Server code

Steps:

6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

RPC: Basic Properties

Basic Properties:

- Synchronous communication
- Only 1 call needed to access remote procedure & get result
  - Other approaches (e.g., IPC) require more
- System takes care of all “small details”
  - Message assembly and disassembly, etc.
- Complexity same as normal procedure call
  - Only one call in progress at a time
- Transparent to distribution
  - As long as client can find a server, it does not matter where it is

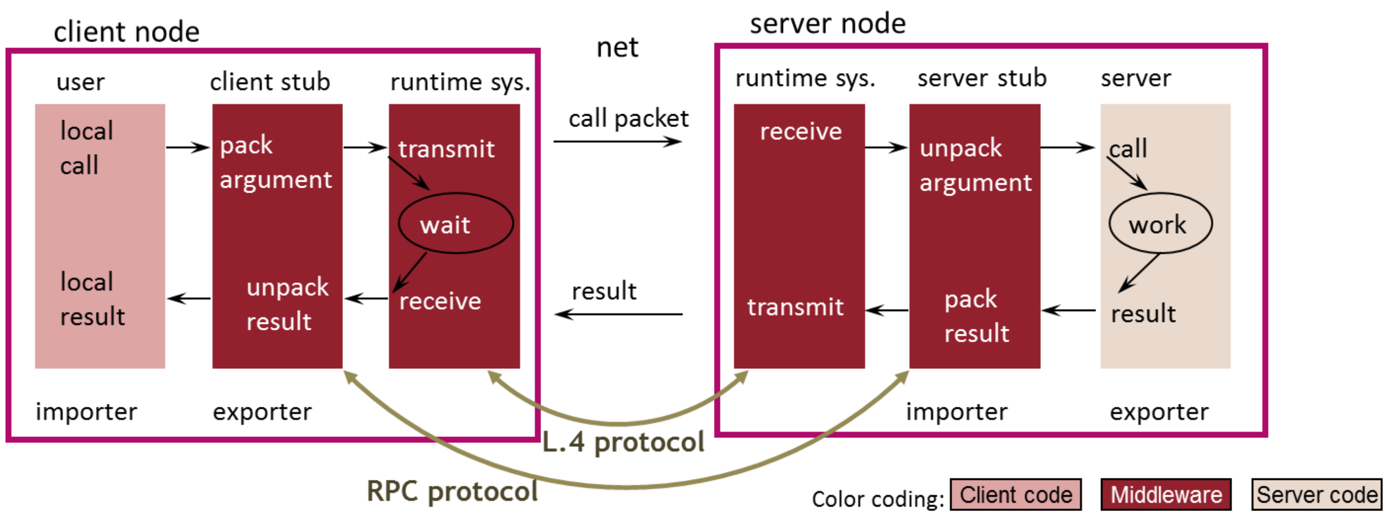
RPC: Basic Functionality

Basic functionality of core RPC middleware:

- Binding: "find appropriate server" (ex-/implicit, may involve *trader*)
- Stub generation (stubs take over most of the work done by IPC programmers before!)
- Within stub & runtime support:
  - Marshalling / Un-marshalling
  - Protocol & error handling → enforcement of "error semantics"
  - Presentation (L.6: translate between heterogeneous OS / prog.lge. / HW)

RPC: Stubs

- Stubs:
  - Mimic local procedure call, hide "networking"
    - Pack / unpack messages (call, reply, ...): Marshalling
    - May convert to / from network data representation
  - Support mapping from client to server: Binding
  - Carry out RPC protocol
- Client stub is proxy for server at client side
- Server stub is proxy for client at server side



RPC: IDLs

- Automatic generation of stub code: stub compiler
- Basis: IDL (Interface definition language)

- Specify the interface between client and server
  - One middleware-specific extension to many programming languages
    - Corba-IDL, DCE-IDL, SUN's XDR (external data rep.), Mach's Matchmaker, XML
  - May use libraries in order to insert code for format conversion, marshalling, transmission control, ...

IDL example:

```
struct Person {
   string name;
   string place;
   long year;
};

interface AddressBook {
   void addPerson(
      in Person p);
   void getPerson(
      in string name,
      out Person p);
   long size();
};
```

RPC: Marshalling and Presentation

- Marshalling challenge: Flatten (serialize) complex data structures
  - Basic data types *plus* structural info

- Accommodate presentation in different OSes, languages, and hardware architectures:
  - Integer (size?, 1's /2's complement?) / float/real (size?, IEEE?, ...)
  - Character (ASCII, unicode?) / string (`\0´ end-flag or byte-count?)
  - Arrays (row- /column-based), struct/union/set... (organization?)
  - Little-endian, big-endian, bit order (MSB→LSB or inverse)

- Worst-case: *n* systems → need ~n^2 conversions
  - Reality: Often 2 possibilities (next slides)

RPC: Marshalling and Presentation

Two possibilities:

1. “Receiver makes it right”
   - Mark representation type
   - Between client and server with same representation, no need for translation (80% of the cases?)
2. Abstract syntax (IDL, or ISO ASN.1) plus standardized network data representation
   - ISO-ASN.1 (abstract syntax notation #1): called BER (basic encoding rules)
   - SUN-XDR (same name as for IDL), adopted by DCE
   - Corba: CDR (Common data representation)
   - Java-RMI: “Java serialized form”
   - XML RPC: XML

Marshalling: CORBA

IDL example:

```
struct Person {
    string name;
    string place;
    long year;
};

interface AddressBook {
    void addPerson(
        in Person p);
    void getPerson(
        in string name,
        out Person p);
    long size();
};
```

Common Data Representation (CDR):

struct with value: {‘Smith’, ‘London’, 1934}

index in sequence of bytes	4 bytes	***notes on representation***
0–3	5	*length of string*
4–7	"Smit"	*‘Smith’*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*‘London’*
20-23	"on__"	
24–27	1934	*unsigned long*

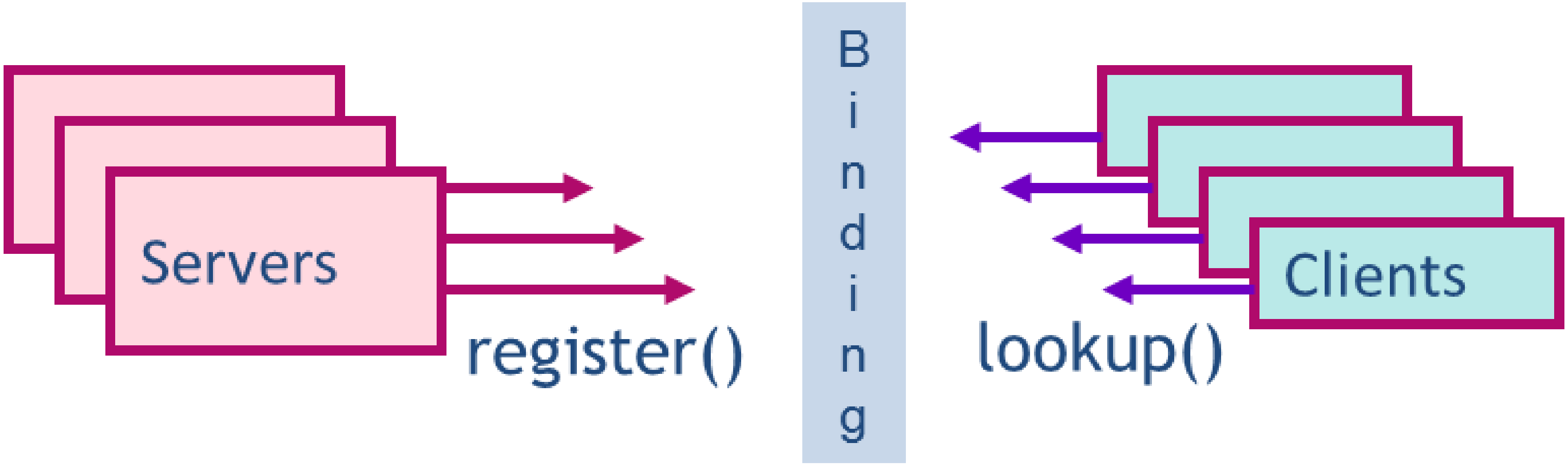
- Primitive types: short…double, char, bool
- Written in sender’s byte order
- Constructed types: sequence, string, array, struct, enumerated, union
- No structural info in CDR, since both sides know from IDL “what comes next” in message

RPC: Binding

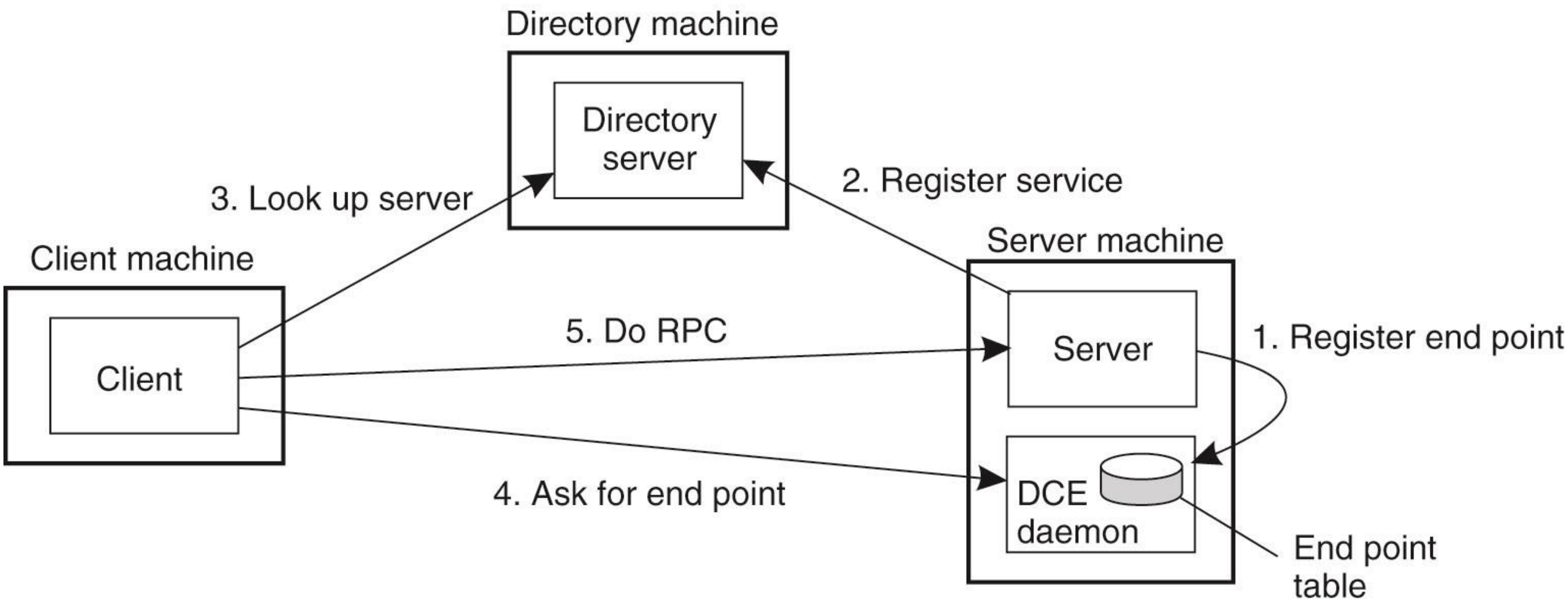
- Binding matches Clients and Servers in 2 steps:
  - Locate the server's machine
  - Locate the server on that machine
- Binding may distinguish just name of server or up to program, version, protocol
- Three possibilities
  1. Static (at compilation time): fast, no middleware overhead
  2. Semi-dynamic (at startup time): logical name/ DB/ multicast/ service
  3. Dynamic (= per call); cf. semi-dynamic, plus: fault tolerance, load balancing

RPC: Binding

- Binding via intermediate service (“trader”, “broker”):
  - cf. “yellow pages” (search via attributes, description) → considered ‘powerful’, but …
    - may become a bottleneck
    - expensive (execution time, triangle communication)



RPC: Binding


Directory machine
Directory server
3. Look up server
2. Register service
Client machine
Server machine
Client
5. Do RPC
Server
1. Register end point
4. Ask for end point
DCE daemon
End point table

RPC: Binding Examples

Binding for Internet-RPC (e.g., based on SUN-RPC / XDR):

- Port mapper rpcbind:
  - Name service for RPC servers on server node: port #111
  - Server registers program & version numbers with local port mapper
  - Check out all registered RPC servers with: rpcinfo –p
- When client calls clnt_create, RPC request is sent to server’s port mapper asking for info for given program, version, & protocol
- RPC request returns server’s port number to client

For Java: “You’re supposed to know your URLs”

For Corba:

- May use naming service to translate logical name
- May use trader service for yellow pages

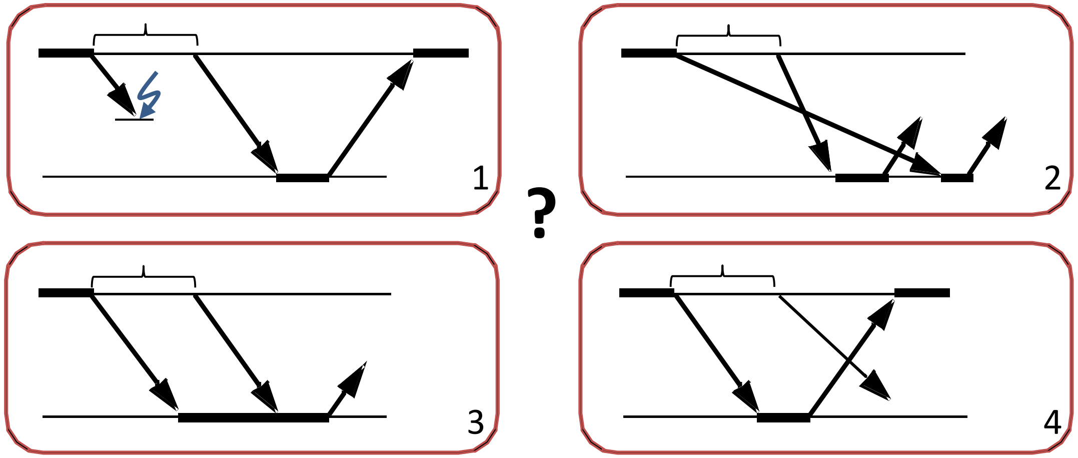
RPC: Protocols

- Must realize three functions:
  - client: do_operation (callee, op_id, *in_args)
  - server: getRequest(...)   and   sendReply (caller, *results)
- Possible errors:
  - Request omission, reply omission (both mean: lost messages)
  - Server crash, client crash
- Error handling:
  - Omissions may be masked, but tradeoffs: „stateful server“ (&client), protocol overhead
  - Server crash commonly not masked (maybe via “dynamic binding”)
- Distinguish two kinds of RPC protocols
  - RR: request-reply
  - RRA: request-reply-ack (ACK by client!)

RPC Errors: Request Omission

Request omission:

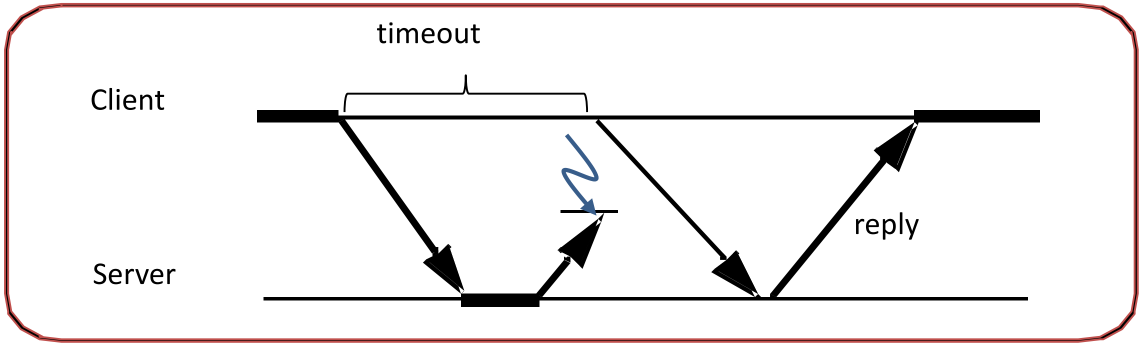
- 1st obvious countermeasure: set timeout!
- In case of timeout → resend request
- However, 4 possible cases:



- Server must recognize duplicate requests → Unique ID for requests
- Server must keep state, length of time determines residual error probability

RPC Errors: Reply Omission

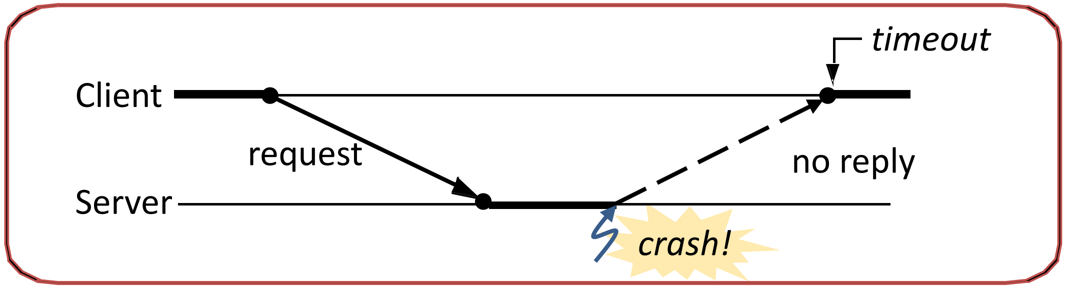
- For client, indistinguishable from request omission & delays
  - Same countermeasure as for request omission (timeout)
- For server, means to memorize results for potential 2nd reply
  - More states, more memory, especially if parallel calls per client
  - Also scalability problem: What if server has thousands of clients?
- Now we see why request-reply-ack (RRA) makes sense
  - If client ACKs reception of reply, server can throw away stored result



RPC Errors: Crashes

Server crash: How far did it get?

- How to distinguish from omissions (*n* timeouts?)
- If other server takes over, how should it know?
- For restart, how to cope with locks, dirty states, info recovery?



Client crash: Server executes "orphan"

- Procedure execution maybe erroneous, "costly", ...
- Restarted client should not be puzzled by orphan-results
  - Might even tell server to stop them?
- Note: For lengthy procedures, server may poll client

RPC Failure Semantics

1. **Maybe** Semantics: No repeated requests (replies, ...)
   - Simple, fast, efficient but often not sufficient
   - Idea: User will try again in case of failure (check email, ...)
2. **At-Least-Once** Semantics: Infinite retry
   - Repeated requests, but stateless servers (no duplicates recognized)
   - Restricted to idempotent operations (basically, “read”-operations)
3. **At-Most-Once** Semantics: Tolerate omissions
   - Repeated requests & replies, duplicates recognized → exec only once
   - Server crash → no result (no reply), server may have executed or not
4. **Exactly-Once** Semantics: Tolerates crashes
   - For normal commercial RPC systems, this remains a dream
   - Transactional systems, fail-safe solutions needed

- Solutions in order of increasing effort
  - Offer the choice to programmer
  - Commercial systems usually offer *some* choice of 1/2/3
- In summary: Forget transparency, accept: RPC ≠ local calls

RPC: Failure Semantics

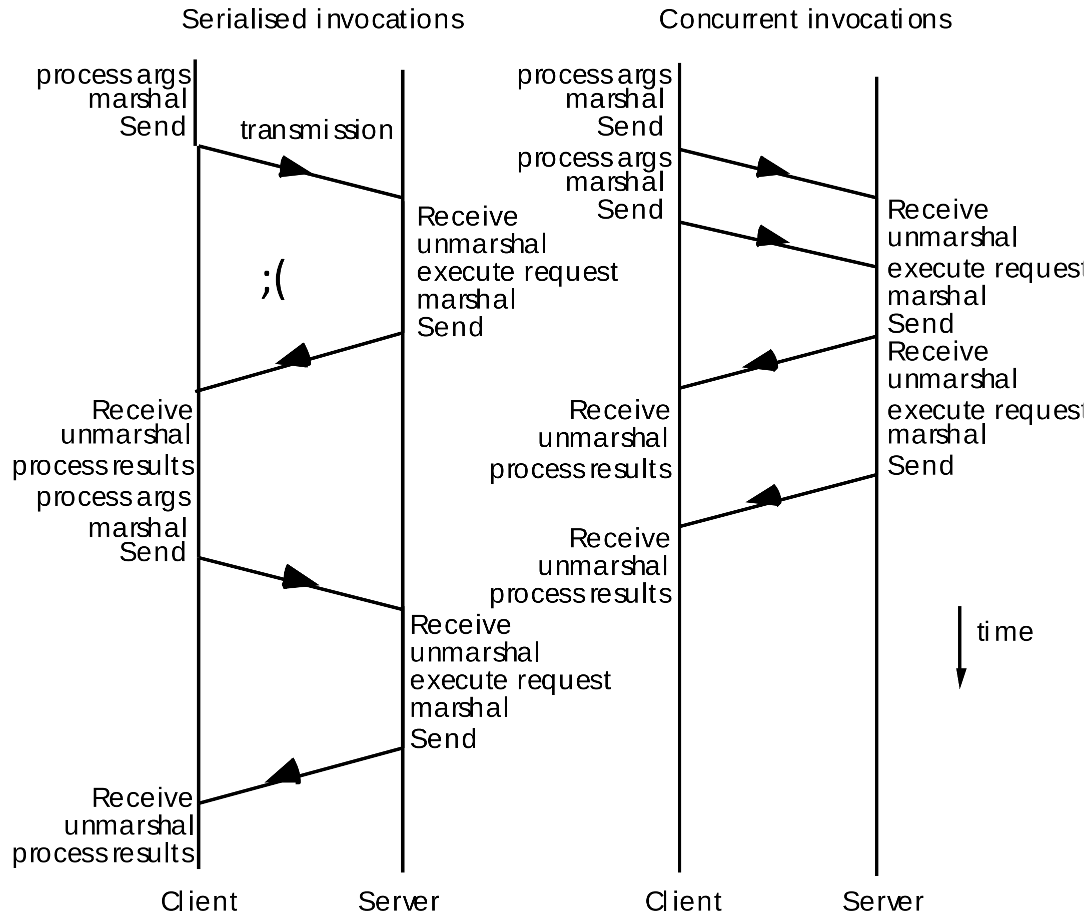
Type of Error Sem.	for absence of errors	in case of omissions	in case of server.crash
maybe	1 proc-exec. 1 result returned	0|1 proc-exec. 0 results returned	0|1 proc-exec. 0 results returned
at-least-once	1 proc-exec. 1 result returned	≥1 proc-exec. ≥1 result returned	≥0 proc-exec. ≥0 result returned
at-most-once	1 proc-exec. 1 result returned	1 proc-exec. 1 result returned	0|1 proc-exec. 0 results returned
exactly-once	1 proc-exec. 1 result returned	1 proc-exec. 1 result returned	1 proc-exec. 1 result returned

- Exactly-once: can be approached via redundancy, but expensive!

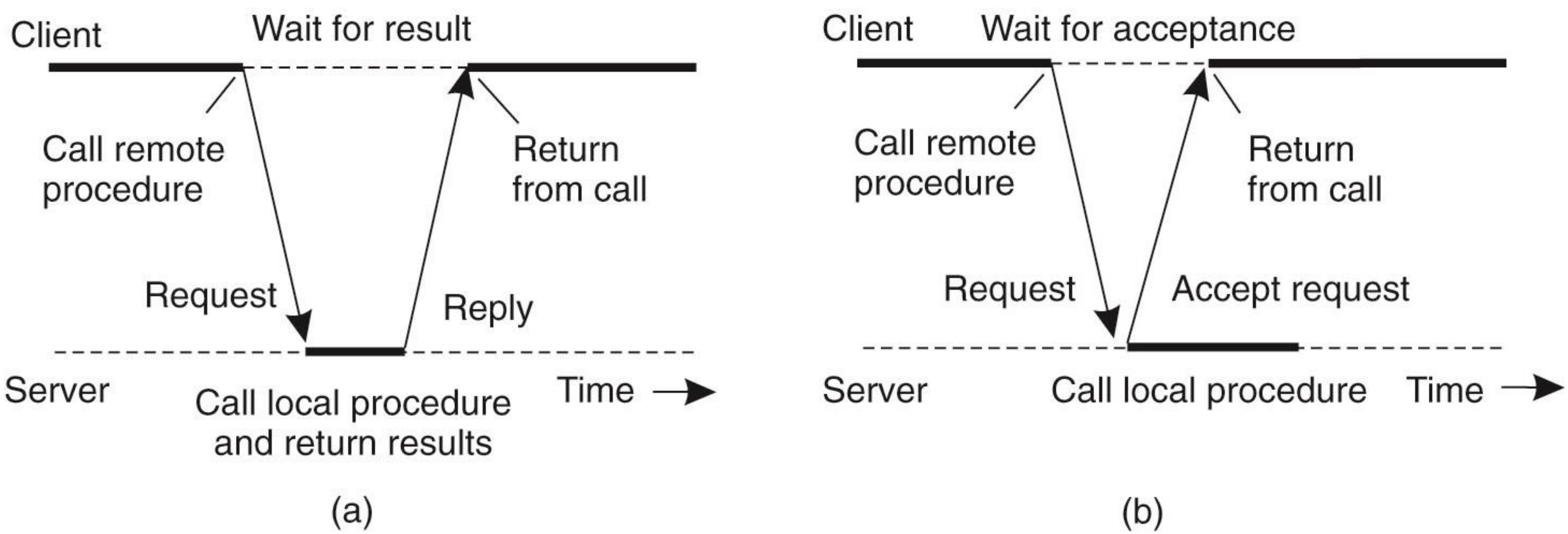
RPC: Asynchronous Calls

- ... the essence of DistSys (e.g., parallel loading of images in http)
- ...and a nightmare for RPC (wrt. transparency, statefulness, ...)

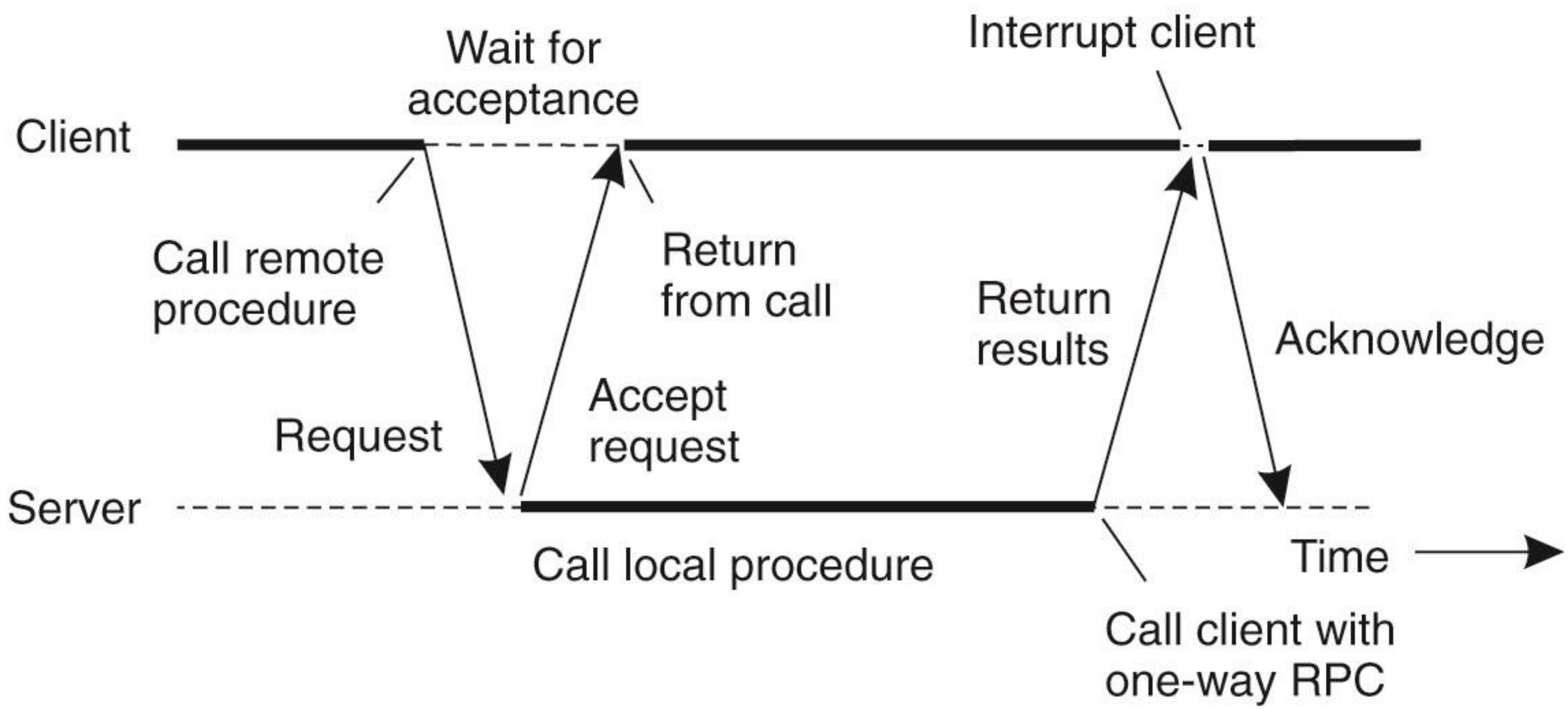
example:
comparison for
just two calls



RPC: Asynchronous Calls

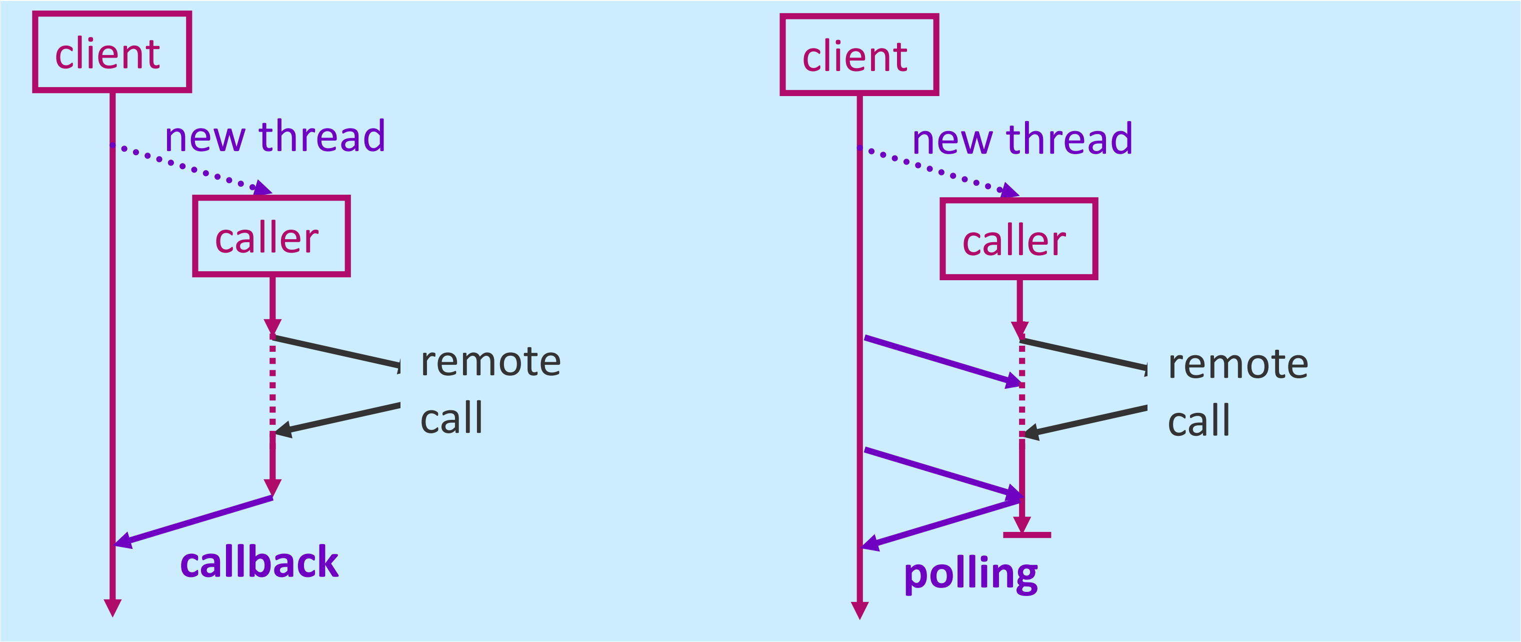


RPC: Asynchronous Calls



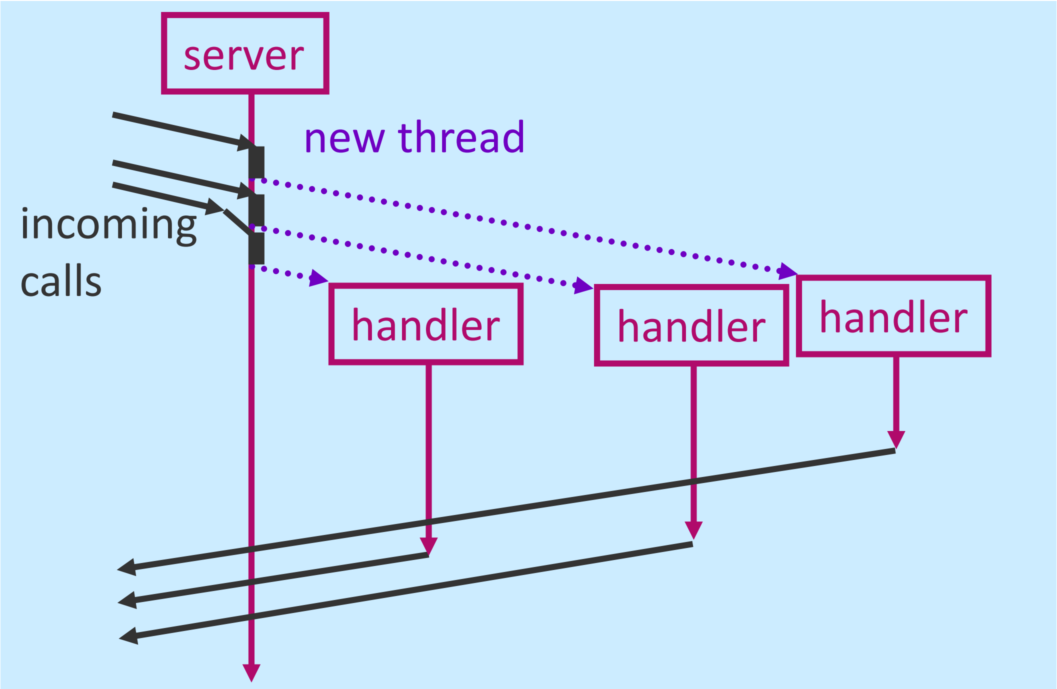
RPC: Asynchronous Calls

- Why multithreading in distributed systems? (the case for RPC)
  - Client side: asynchronous calls
  - Server side: parallel handlers
- Asynchronous Call
  - Split off caller thread
  - Result obtained by callback or polling



RPC: Parallel Handlers

- Multithreading on server side: parallel handlers



Inlet: Concurrency

TECHNISCHE UNIVERSITÄT DARMSTADT

What?

- Traditionally, concurrency was issue of OS
- Became issue of concurrent languages via „threads“
  - Thread: light-weight process
  - Has own registers & stack, but not own address space (well, maybe TLS)
  - Shorter creation time
  - Threads in same process / on the same VM allow for more efficient synchronization concepts
  - Reduced „security“ (same user address space)

Why?

1. Parallelism essential for DistSys
   - DistSys is inherently parallel
   - E.g., we'll receive RPC requests in parallel to the main control flow
2. Concurrency models to be „inherited“ into DistSys?
   - Yes: Monitor, Semaphor, Lock “& friends”

Concurrency: Problems

- **Deadlock**
  - A condition that occurs when two processes are waiting for each other to complete before proceeding. The result is that both processes hang.
  - Classical example:
    - P1: lock(X); lock(Y); … unlock(Y); unlock(X);
    - P2: lock(Y); lock(X); … unlock(X); unlock(Y);
  - (Solution for classical example: define order for locks)
- **Livelock / Starvation**
  - A condition that occurs when two or more processes continually change their state in response to changes in the other processes. The result is that none of the processes will complete.
- **Unfairness**
  - Fairness is related to scheduling and concerned with guaranteeing the processes get a chance to proceed (‘sufficiently fast, sufficiently often’)
- **Race Condition**
  - A race condition, here, is defined to be a timing-related flaw in a system: the result (/completion) of a computation is *unexpectedly* and *critically* dependent on the sequence or timing of events.
  - Race conditions arise not because of message delay per se, but because of **varying** processing/scheduling/transmission timings.

Concurrency: Synchronization

- **Monitor** (according to Tanenbaum; terms in literature not 100% consistent)
  - Programming-language construct
  - A monitor is a „module“ containing variables and procedures
  - Variables can only be accessed via procedures (data encapsulation)
  - If process A executes a procedure (enters the monitor), then a process B trying to execute a procedure of the same monitor, will be blocked until A exits.
  - Every Java object has a built-in monitor lock: use **synchronized**, e.g.:

```
public class CountingIntMonitor {
    private int value;
    public synchronized int value()  { return value; }
    public synchronized void increment() { value = value + 1; }
}
```

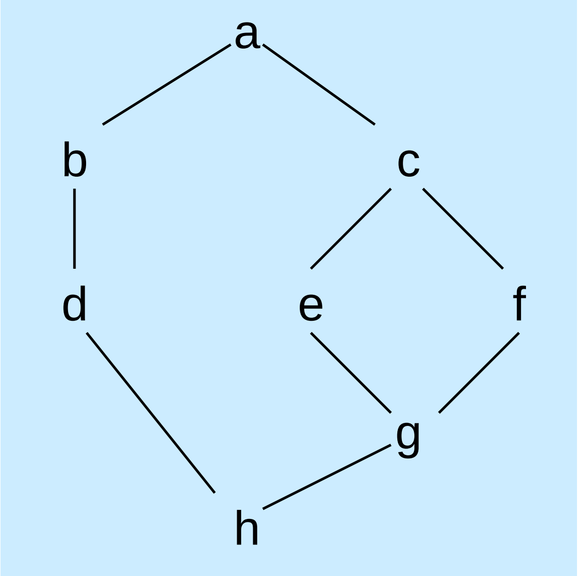
- **Synchronized** has two effects:
  - Concurrent invocations **cannot interleave**
  - Establishes a **happened-before relationship** with any subsequent call
    - guarantees that changes to object state are visible to all threads

Block-level Concurrency

- Finer-grained concept than Threads
- Use block objects and execution queues instead of Threads
  - reduced memory penalty: blocks don't have separate stack (→ no recursion!)
  - less overhead: No thread creation / cleanup / synchronization; simpler scheduling
- Examples
  - Apple Grand Central dispatch
    - Support for block objects in C, C++, and ObjC
    - Unified dispatch method for CPUs and GPUs
  - Microsoft .NET Task Parallel Library (TPL)
    - Support for Action objects = lambda expressions

```
a();
Parallel.Invoke(
   () => { b(); d(); },
   () => { c(); Parallel.Invoke(
      () => e(),
      () => f()); g();
   });
h();
```

Path Expressions: explicit notation for sequential / parallel execution



RPC: Known Issues

RPC is not equal to local procedure call:

- Vast area of error semantics & error control vs. all-or-nothing
- Separate addr.spaces → how are parameters/return values passed?
  - Call by value/copy .. /reference?
  - Request/reply vs. request/reply/restore
  - Serialization, ability to „transmit“ complex data structures (list? tree? w/ ref’s?)
  - (usually) no support for variable parameter lists
  - (usually) no support for pointers as parameters
- Lack of shared variables (→ of side effects)
- Lack of performance transparency
  - WANs: extremely long and varying response times
  - Performance degradation via indirections (traders, ORBs, software busses, etc.)
  - Overhead for marshalling, serialization, etc.
- Problems w/ mass data, multimedia
- Security aspects
  - Integrate authentication, authorization, key exchange w/ binding?

RPC: Summary

- Powerful tool for medium sized problem
- "Does the plumbing for you"
  - Stubs, Marshalling, Binding etc.
  - Good libraries for most languages
- Difference between local and remote calls
- Apache Thrift (Facebook), Protocol Buffers (Google), Finagle (Twitter)
- Support for high level of concurrency
  - Asynchronous by design
  - Lightweight threads, e.g., Futures etc.
  - Small protocol overhead