Communication in Distributed Systems

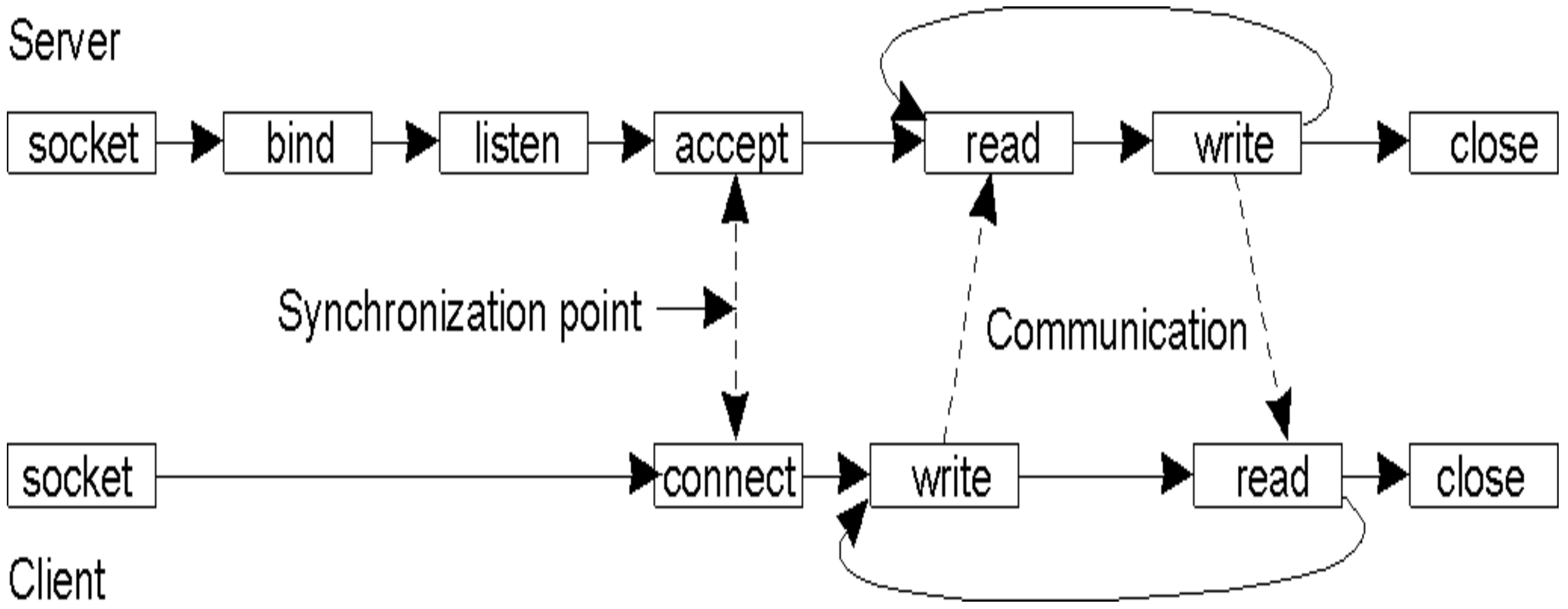
Review

- In a distributed system, processes
  - run on different machines
  - exchange information through message passing
- Successful distributed systems depend on communication models that hide or simplify message passing

Middleware Communication Techniques

- Message-Oriented Communication
- Remote Procedure Call
- Stream-Oriented Communication

Message-oriented Transient Communication Berkley Sockets



Berkeley Socket Primitives

Primitive	**Meaning**
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Message-Passing Interface (MPI)

- Sockets designed for network communication (e.g., TCP/IP)
  - Support simple send/receive primitives
- Abstraction not suitable for other protocols in clusters of workstations or massively parallel systems
  - Need an interface with more advanced primitives
- Large number of incompatible proprietary libraries and protocols
  - Need for a standard interface
- Message-passing interface (MPI)
  - Hardware independent
  - Designed for parallel applications
- Key idea: communication between groups of processes
  - Each endpoint is a *(groupID, processID)* pair

MPI Primitives

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there are none
MPI_irecv	Check if there is an incoming message, but do not block

Remote Procedure Call

Motivation

- The sockets interface forces a read/write mechanism
- Programming is often easier with a functional interface
- To make distributed computing look more like centralized computing, I/O (read/write) is not the way to go

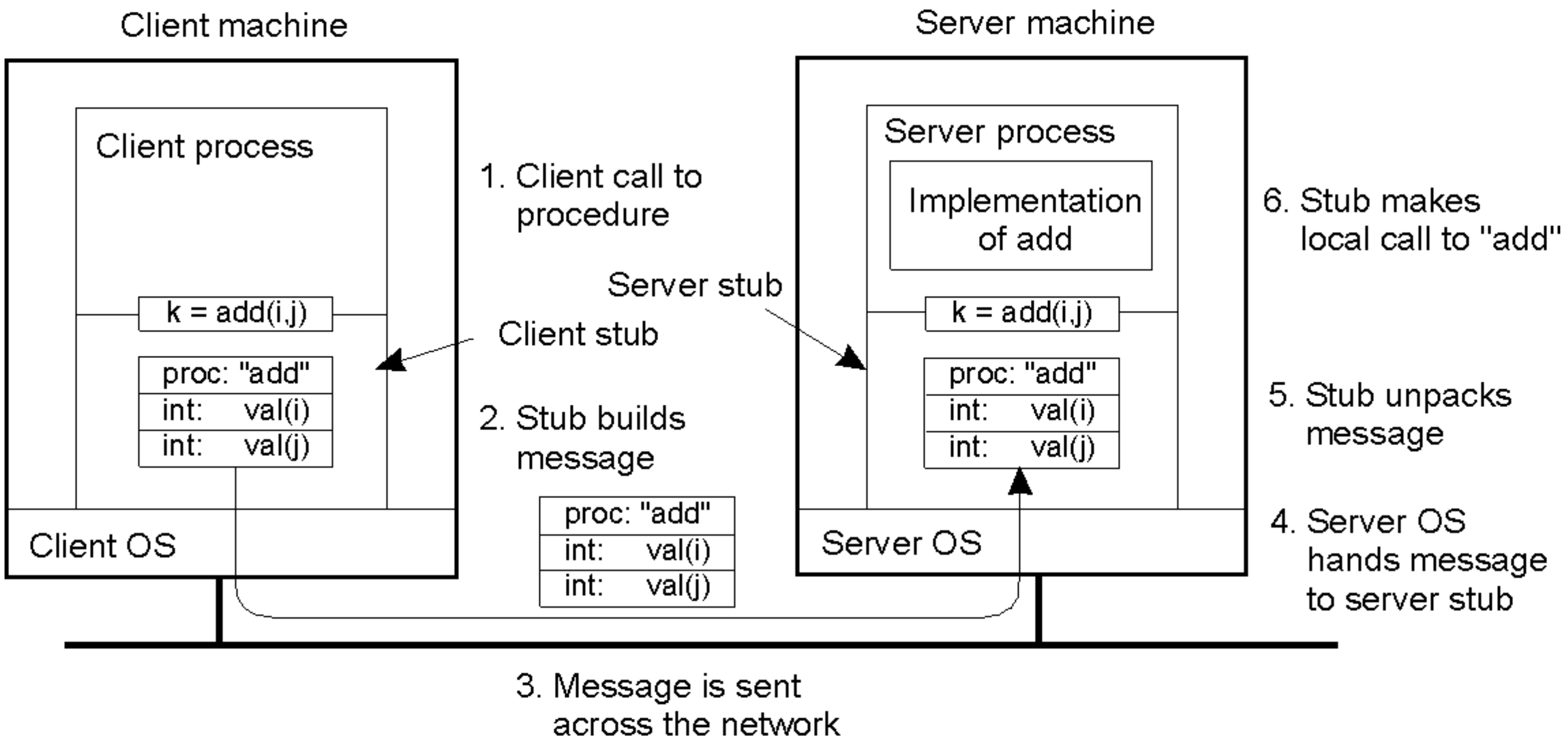
Remote Procedure Calls

- Client/server model
- Request/reply paradigm usually implemented with message passing in RPC service
- Marshalling of function parameters and return value

Remote Procedure Calls

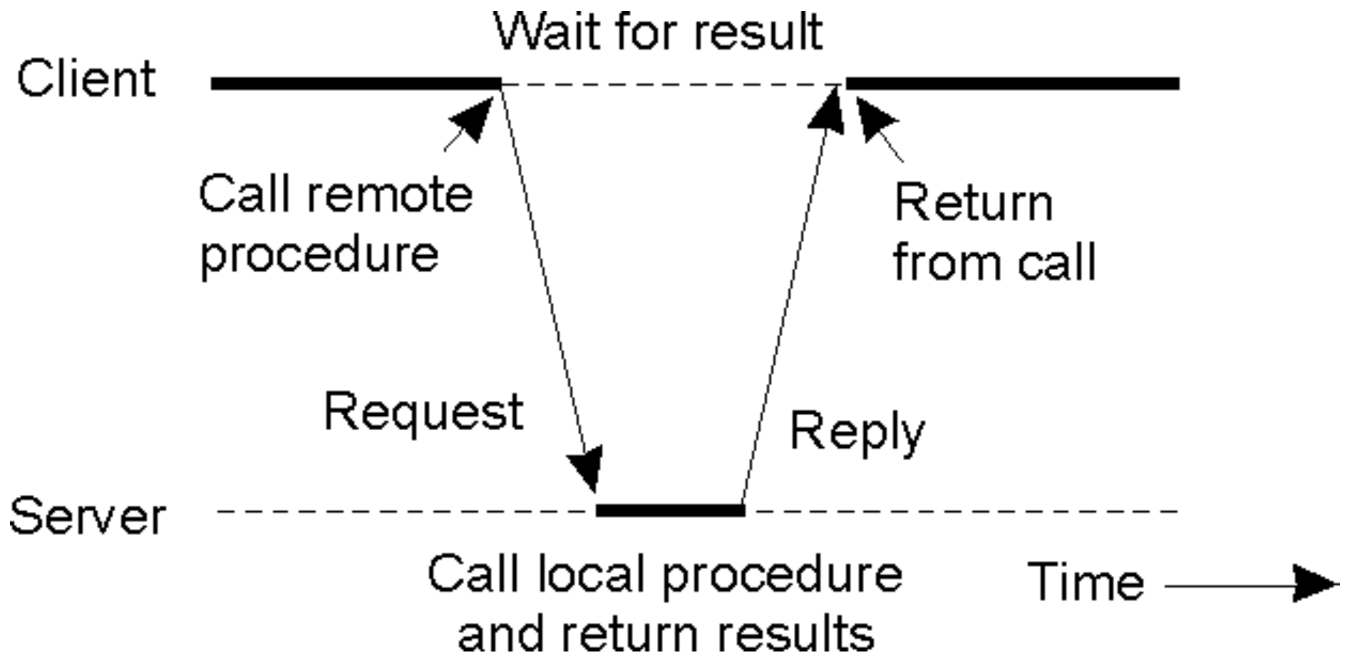
- Goal: Make distributed computing look like centralized computing
- Allow remote services to be called as procedures
  - Transparency with regard to location, implementation, language
- Issues
  - How to pass parameters
  - Bindings
  - Semantics in face of errors
- Two classes: integrated into prog language and separate

Example of an RPC



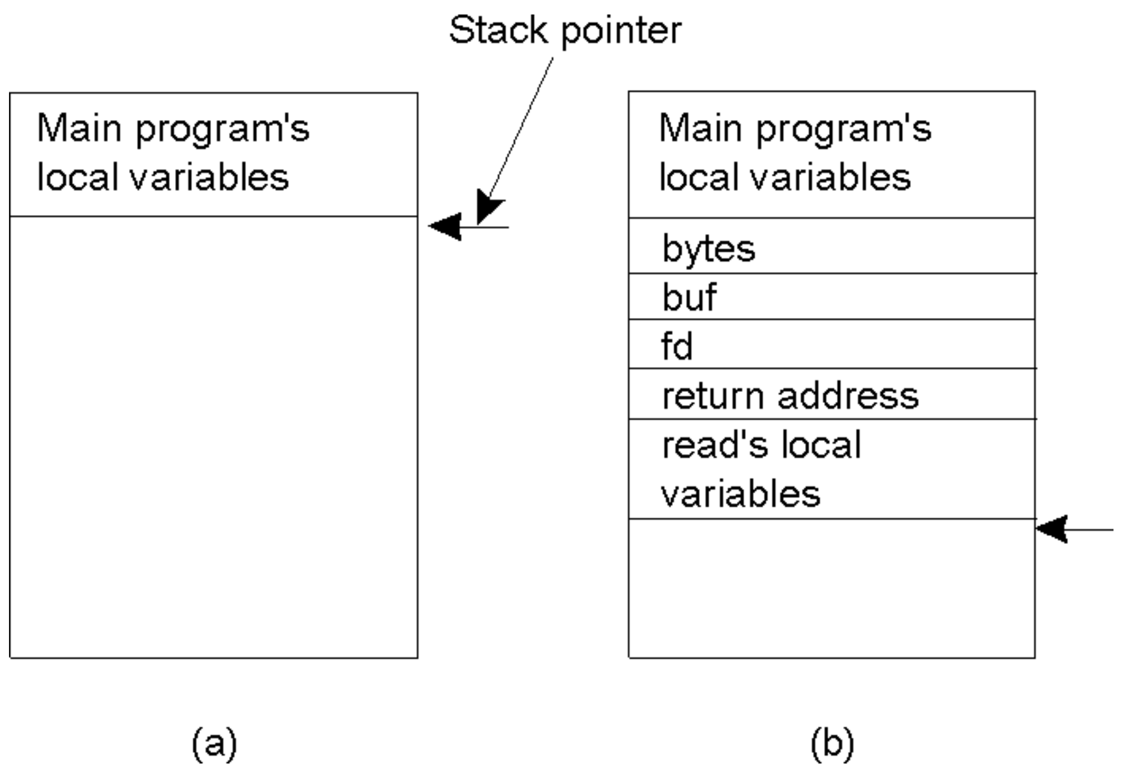
RPC Semantics

- Principle of RPC between a client and server program [Birrell&Nelson 1984]



Conventional Procedure Call

a) Parameter passing in a local procedure call: the stack before the call to read b) The stack while the called procedure is active



Parameter Passing

- Local procedure parameter passing
  - Call-by-value
  - Call-by-reference: arrays, complex data structures
- Remote procedure calls simulate this through:
  - Stubs – proxies
  - Flattening – marshalling
- Related issue: global variables are not allowed in RPCs

Client and Server Stubs

- Client makes procedure call (just like a local procedure call) to the client stub
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages
- Packaging parameters is called *marshalling*
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
  - Simplifies programmer task

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Marshalling

- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
  - ASCII vs. UNICODE
- Solution: use a standard representation
  - Example: external data representation (XDR)
- Problem: how do we pass pointers?
  - If it points to a well-defined data structure, pass a copy and the server stub passes a pointer to the local copy
- What about data structures containing pointers?
  - Prohibit
  - Chase pointers over network
- Marshalling: transform parameters/results into a byte stream

Serialization

- Need standard encoding to enable communication between heterogeneous systems
- Serialization
  - Convert data into a pointerless format: an array of bytes
- Examples
  - XDR (eXternal Data Representation), used by ONC RPC
  - JSON (JavaScript Object Notation)
  - W3C XML Schema Language
  - ASN.1 (ISO Abstract Syntax Notation)
  - Google Protocol Buffers

Serialization

- Implicit typing
  - only values are transmitted, not data types or parameter info
  - e.g., ONC XDR (RFC 4506)
- Explicit typing
  - Type is transmitted with each value
  - e.g., ISO's ASN.1, XML, protocol buffers, JSON

Marshaling vs. Serialization

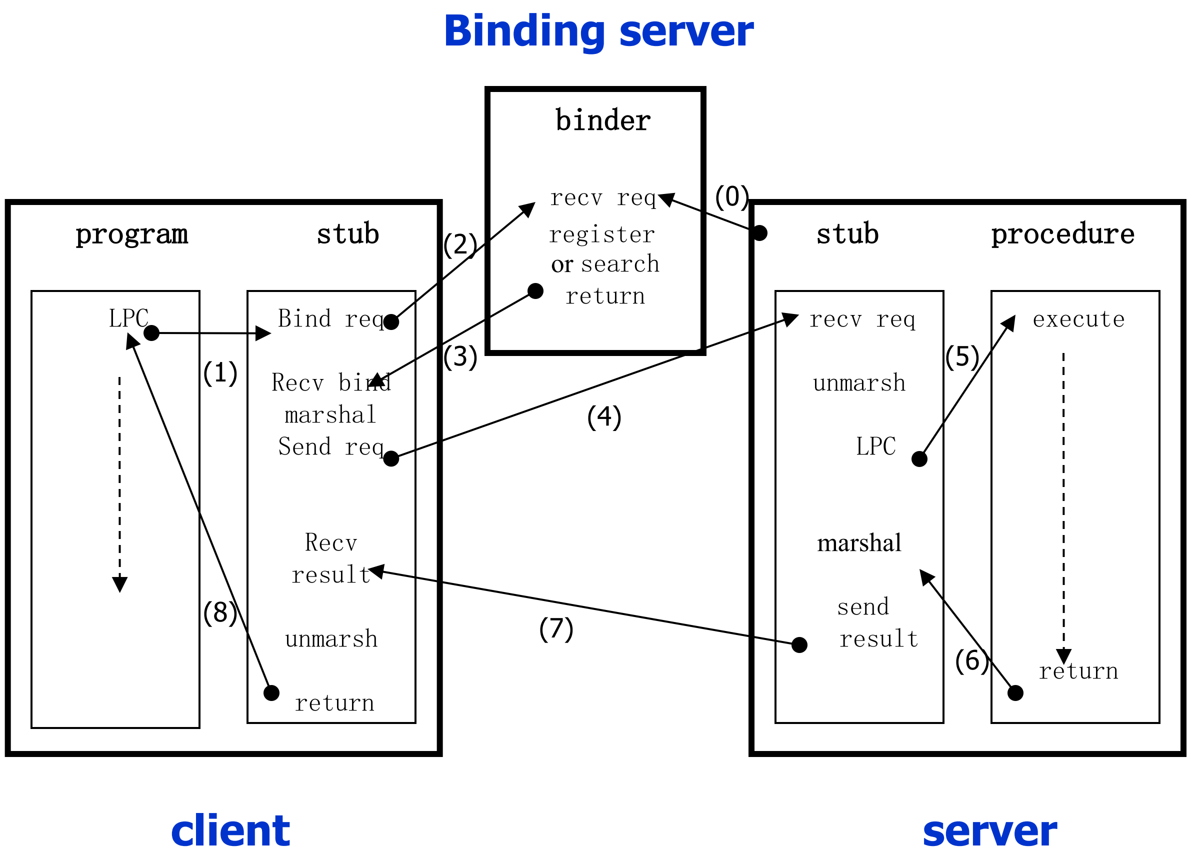
- **Serialization**: converting an object into a sequence of bytes that can be sent over a network

- **Marshaling**: bundling parameters into a form that can be reconstructed (unmarshaled) by another process. May include object ID or other state. Marshaling uses serialization.

Binding

- Problem: how does a client locate a server?
  - Use Bindings (Static vs. Dynamic)
- Server
  - Export server interface during initialization
  - Send name, version no, unique identifier, handle (address) to binder
- Client
  - First RPC: send message to binder to import server interface
  - Binder: check to see if server has exported interface
    - Return handle and unique identifier to client

Dynamic Binding



Binding: Comments

- Exporting and importing incurs overheads
- Binder can be a bottleneck
  - Use multiple binders
- Binder can do load balancing

RPC Implementation
Failure Semantics

- ***Client unable to locate server***: return error
- ***Lost request messages***: simple timeout mechanisms
- ***Lost replies***: timeout mechanisms
  - Make operation idempotent
  - Use sequence numbers, mark retransmissions
- ***Server failures****:* did failure occur before or after operation?
  - At least once semantics (SUNRPC)
  - At most once
  - No guarantee
  - Exactly once: desirable but difficult to achieve

Failure Semantics

- ***Client failure**: what happens to the server computation?*
  - Referred to as an *orphan*
  - *Extermination*: log at client stub and explicitly kill orphans
    - Overhead of maintaining disk logs
  - *Reincarnation*: Divide time into epochs between failures and delete computations from old epochs
  - *Gentle reincarnation*: upon a new epoch broadcast, try to locate owner first (delete only if no owner)
  - *Expiration*: give each RPC a fixed quantum *T*; explicitly request extensions
    - Periodic checks with client during long computations

Implementation

- Many programming languages have no language-level concept of remote procedure calls (C, C++, Java < J2SE 5.0)
  - These compilers will not automatically generate client and server stubs (so we use special tools/generators).
- Some languages have support that enables RPC (Java, Python, Haskell, Go, Erlang)
  - But we may need to deal with heterogeneous environments (e.g., Java communicating with a Python service)
- Common solution
  - Interface Definition Language (IDL): describes remote procedures
  - Separate compiler that generate stubs (pre-compiler)

Interface Definition Language (IDL)

- Allow programmer to specify remote procedure interfaces (names, parameters, return values)
- Pre-compiler can use this to generate client and server stubs –
  - Marshaling code
  - Unmarshaling code
  - Network transport routines
  - Conform to defined interface
- An IDL looks similar to function prototypes

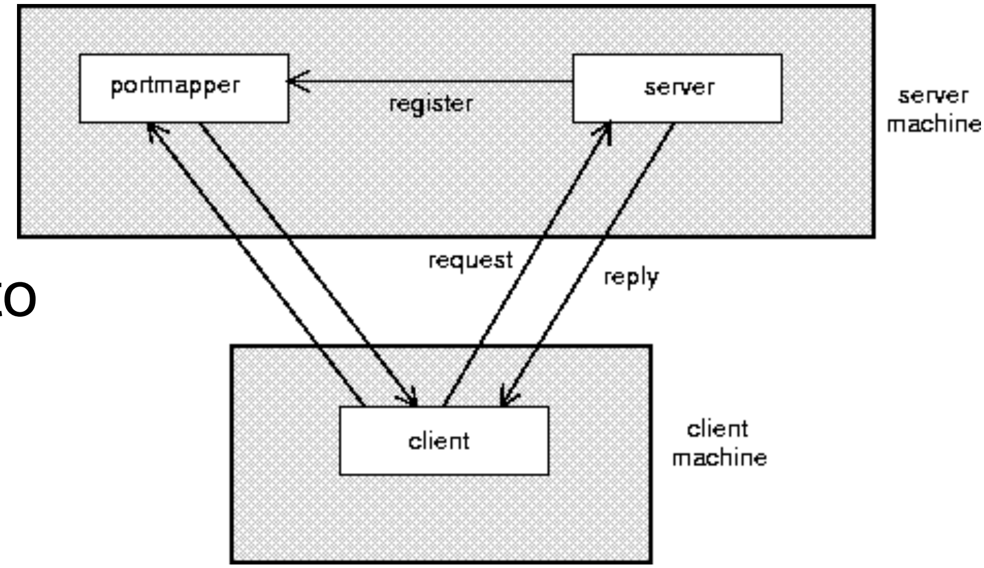
Writing the program

- Client code has to be modified
  - Initialize RPC-related options •
    - Identify transport type •
    - Locate server/service
  - Handle failure of remote procedure calls •
- Server functions
  - Generally need little or no modification

Case Study: SUNRPC

- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
  - TCP: stream is divided into records
  - UDP: max packet size < 8912 bytes
  - UDP: timeout plus limited number of retransmissions
  - TCP: return error if connection is terminated by server
- Multiple arguments marshalled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN's eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers, handle arbitrarily large data structures

Binder: Port Mapper



- Server start-up: create port
- Server stub calls *svc_register* to register prog. #, version # with local port mapper
- Port mapper (binder) stores prog #, version #, and port
- Client start-up: call *clnt_create* to locate server port
- Upon return, client can call procedures at the server

Original Program

```
main()
{
  char cip[] = "Buubdl!bu!ebxo";  /* cipher*/
  int key = 1;/* secret  key */
  int len = decrypt(cip, key);         /* LPC  */
      /*  other processing */
}

int decrypt(char * s, int key)
{           /* decryption  */
   int i = 0;
   while( *s) { *s -= key; i++; s++;}
   return i;
}

int encrypt(char * s, int key)
{ … }
```

Example: SUN RPC (1)

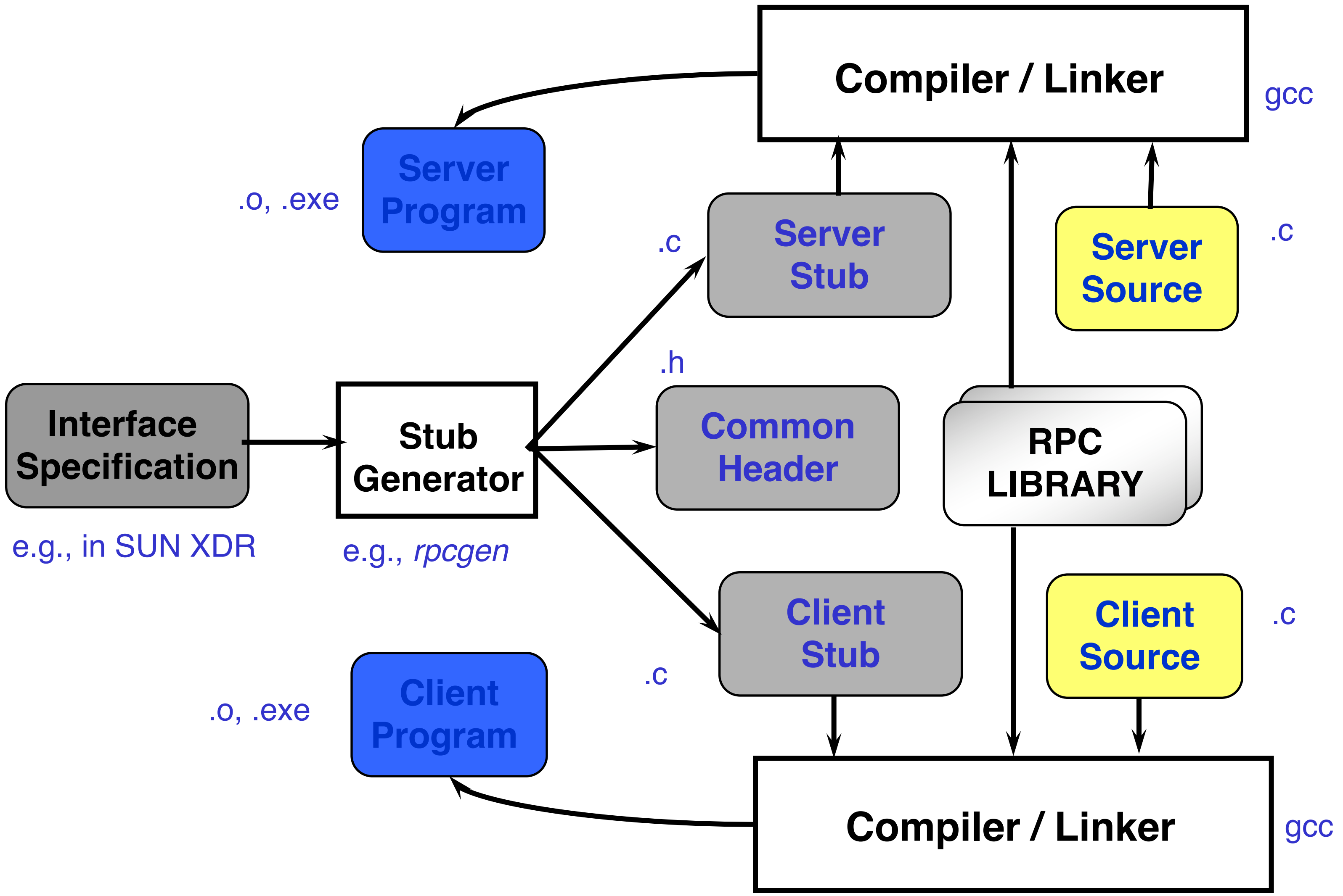
```
/* IDL File : caesar.x */

const MAX = 100;
typedef struct {                    /* return type */
        int len;
        char code[MAX];
} Data;

typedef struct {                    /* parameter type */
        int key;
        char cipher[MAX];
} Args;

program CAESAR {        /* CAESAR program */
    version VERSION {
            Data DECRYPT(Args) = 1; /* decryption procedure */
            Data ENCRYPT(Args) = 2; /* encryption procedure*/
        } = 5;
} = 0x88888888;
```

Example: SUN RPC(3)

```
/* client program file:  client.c */
#include <rpc/rpc.h>
#include "caesar.h"

main(){
  CLIENT *cp;
  char *serverName = "Caesar_server";
  Args arg;
  Data * plaintext;
  /* create client pointer */
  cp = clnt_create(serverName, CAESAR,
VERSION, "udp");
  if (cp == NULL)  exit(1);
  arg.key = 1;                    /* set RPC
parameters  */
  arg.cipher = "Buubdl!bu!ebxo";
  plaintext = decrypt_2(&arg, cp); /*issue RPC*/

  /* other processing */
  ...
  clnt_destroy(cp);/* delete client pointer  */
}
```

```
/* server program file:  server.c */
#include <rpc/rpc.h>
#include "ceasar.h"

Data* decrypt_2(Args *a){ /*
decryption  */
  static Data output;  /* must be
static  */
  char s = a->cipher;
  int i = 0;
  while( *s) { output.code[i] = *s - key;
i++; s++;}
                output.len = i;
  return &output;  /* return result
*/
}

Data* encrypt_2(args *a){ /*
encryption */
 /* ...  */
}
```

The Stub Generation Process



Disadvantages of RPC

- ✖ Synchronous request/reply interaction
  - tight coupling between client and server
  - client may block for a long time if server loaded
  - slow/failed clients may delay servers when replying
- ✖ Distribution Transparency
  - Not possible to mask all problems
- ✖ RPC paradigm is not object-oriented
  - invoke functions on servers as opposed to methods on objects

DCE RPC

- Improved SUN RPC
- Interfaces => Interface Definition Notation (IDN)
  - Definitions look like function prototypes
- Run-time libraries
  - One for TCP/IP and one for UDP/IP
- Authenticated RPC support with DCE security services
- Integration with DCE directory services to locate servers
  - No need to know which machine provides a service

Distributed Component Object (DCOM)

- DCOM specifies the Distributed Component Object Model (DCOM) Remote Protocol
  - Exposes application objects via remote procedure calls (RPCs)
  - Consists of a set of extensions layered on the Microsoft Remote Procedure Call Extensions.
- The DCOM Remote Protocol is also referred to as Object RPC or ORPC.
  - A remote procedure call whose target is an interface on an object.
- ORPC is a small extension of the DCE RPC protocol
- DCOM is used in MS SAMBA File Server.

Lightweight RPCs

- Many RPCs occur between client and server on same machine
  - Need to optimize RPCs for this special case => use a lightweight RPC mechanism (LRPC)
- Server *S* exports interface to remote procedures
- Client *C* on same machine imports interface
- OS kernel creates data structures including an argument stack shared between *S* and *C*

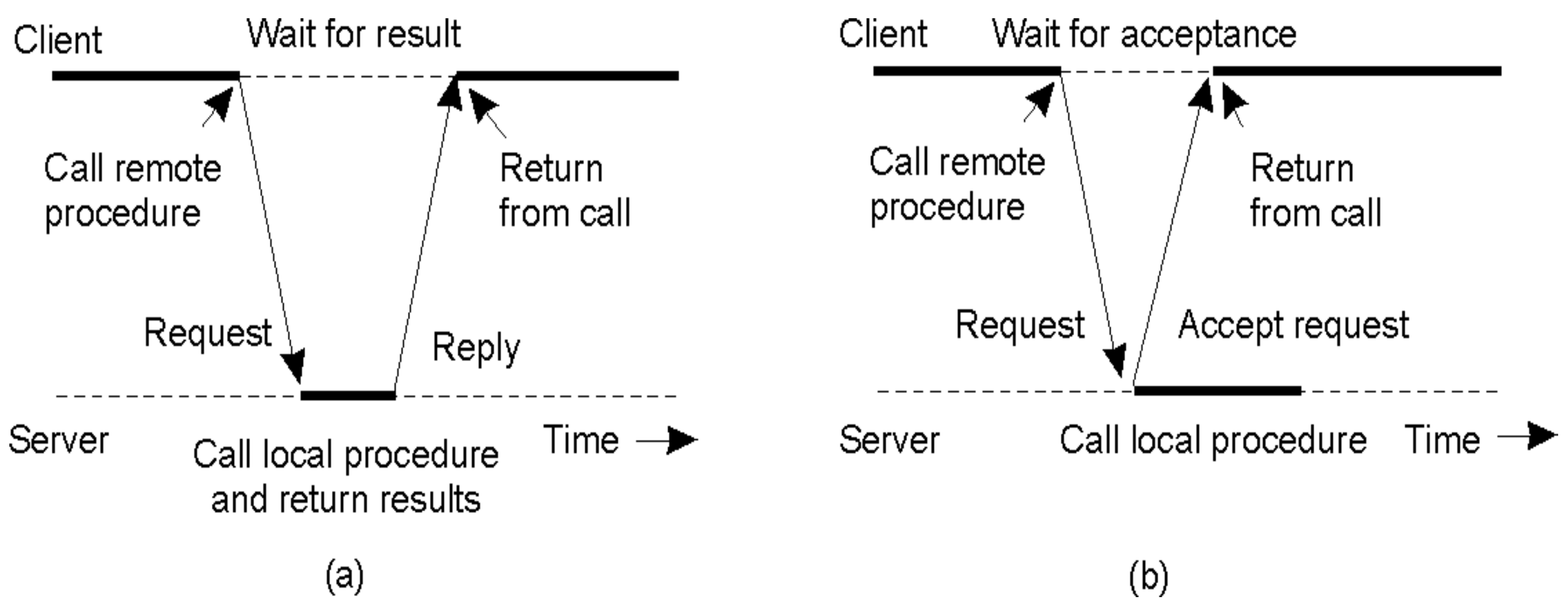
Lightweight RPCs

- RPC execution
  - Push arguments onto stack
  - Trap to kernel
  - Kernel changes mem map of client to server address space
  - Client thread executes procedure (OS upcall)
  - Thread traps to kernel upon completion
  - Kernel changes the address space back and returns control to client
- Called “doors” in Solaris

Other RPC Models

- Asynchronous RPC
  - Request-reply behavior often not needed
  - Server can reply as soon as request is received and execute procedure later
- Deferred-synchronous RPC
  - Use two asynchronous RPCs
  - Client needs a reply but can't wait for it; server sends reply via another asynchronous RPC
- One-way RPC
  - Client does not even wait for an ACK from the server
  - Limitation: reliability not guaranteed (Client does not know if procedure was executed by the server).

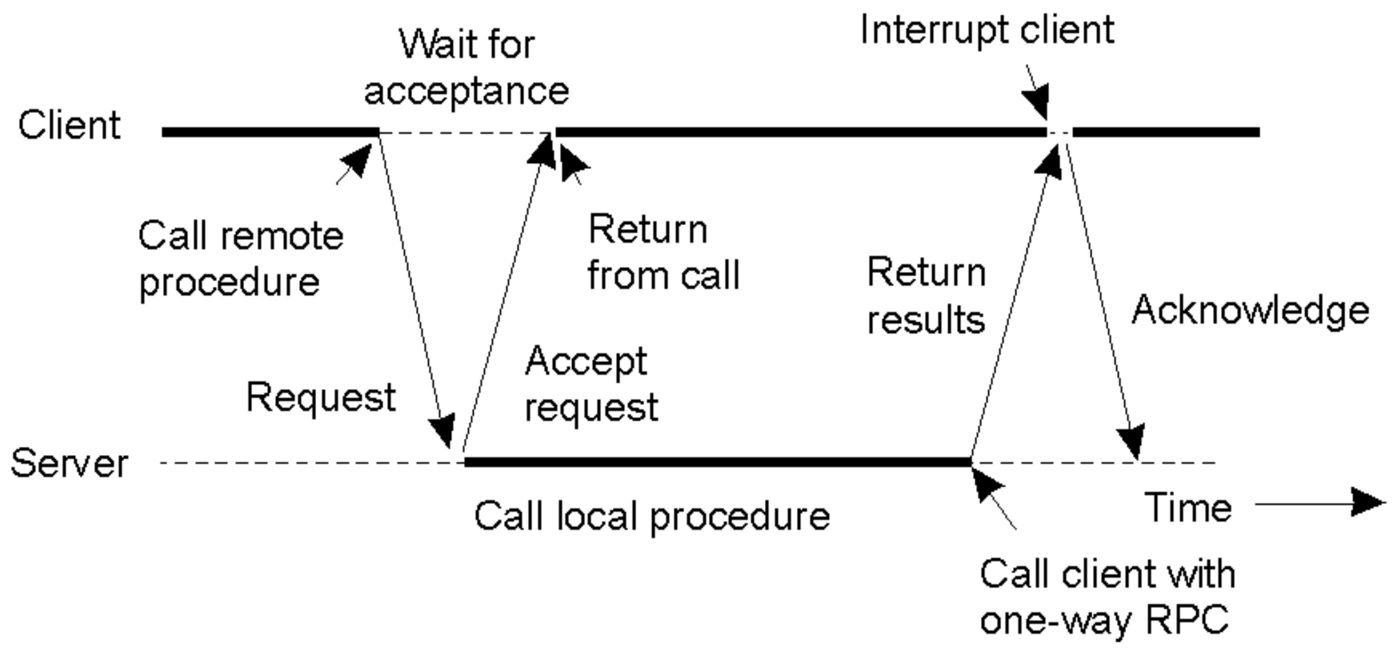
Asynchronous RPC



a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC

Deferred Synchronous RPC

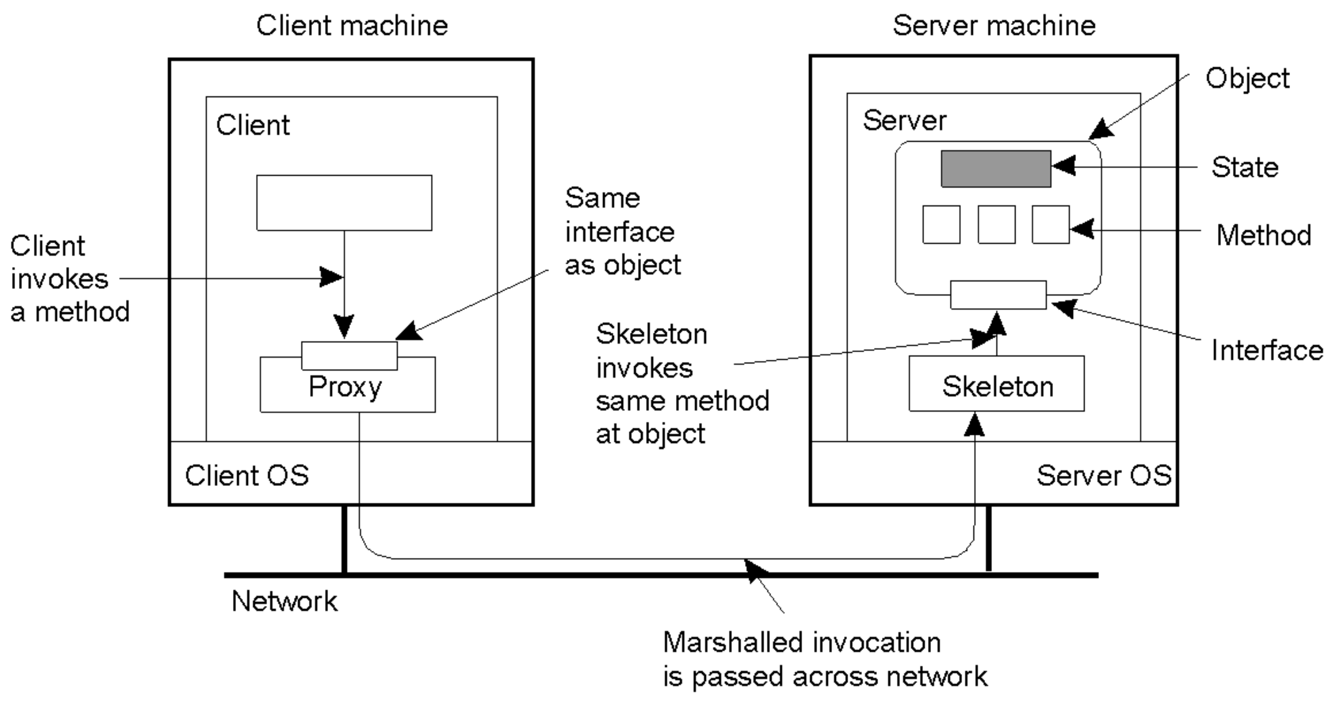
A client and server interacting through two asynchronous RPCs



Remote Method Invocation (RMI)

- RPCs applied to *objects,* i.e., instances of a class
  - *Class:* object-oriented abstraction; module with data and operations
  - Separation between interface and implementation
  - Interface resides on one machine, implementation on another
- RMIs support system-wide object references
  - Parameters can be object references

Distributed Objects



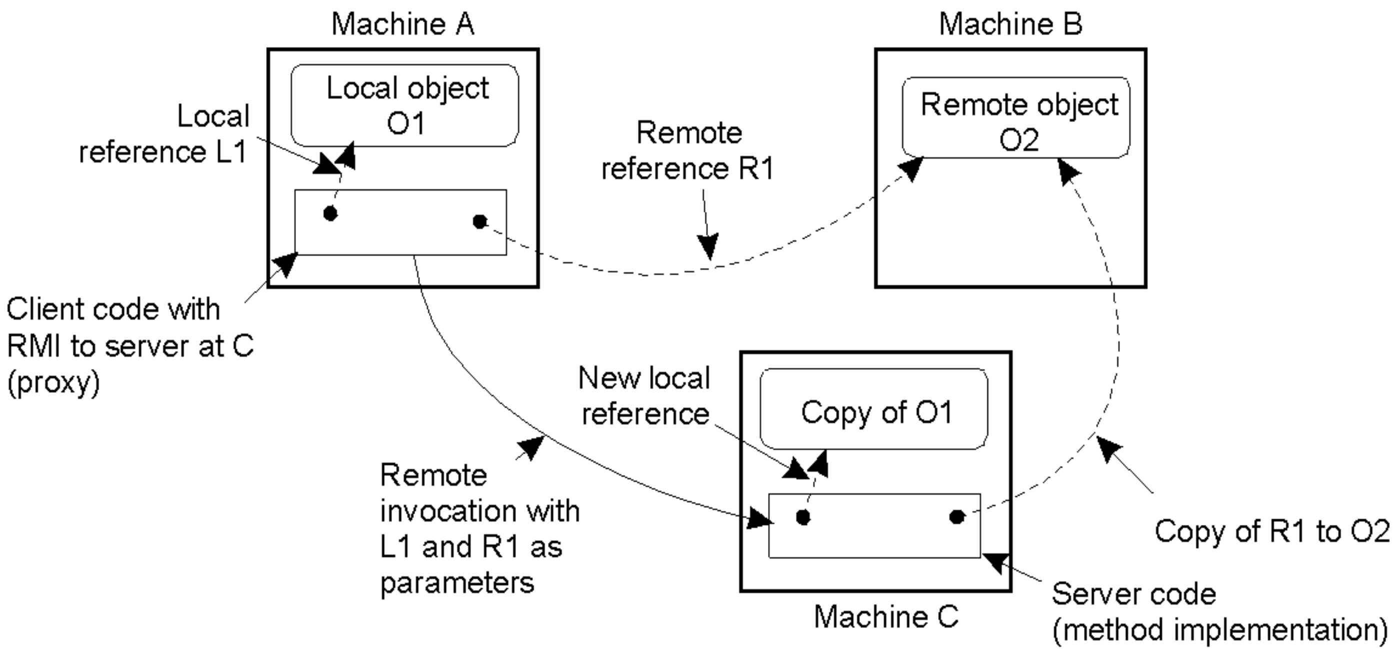
- When a client binds to a distributed object, load the interface ("proxy") into client address space
  - Proxy analogous to stubs
- Server stub is referred to as a skeleton

Proxies and Skeletons

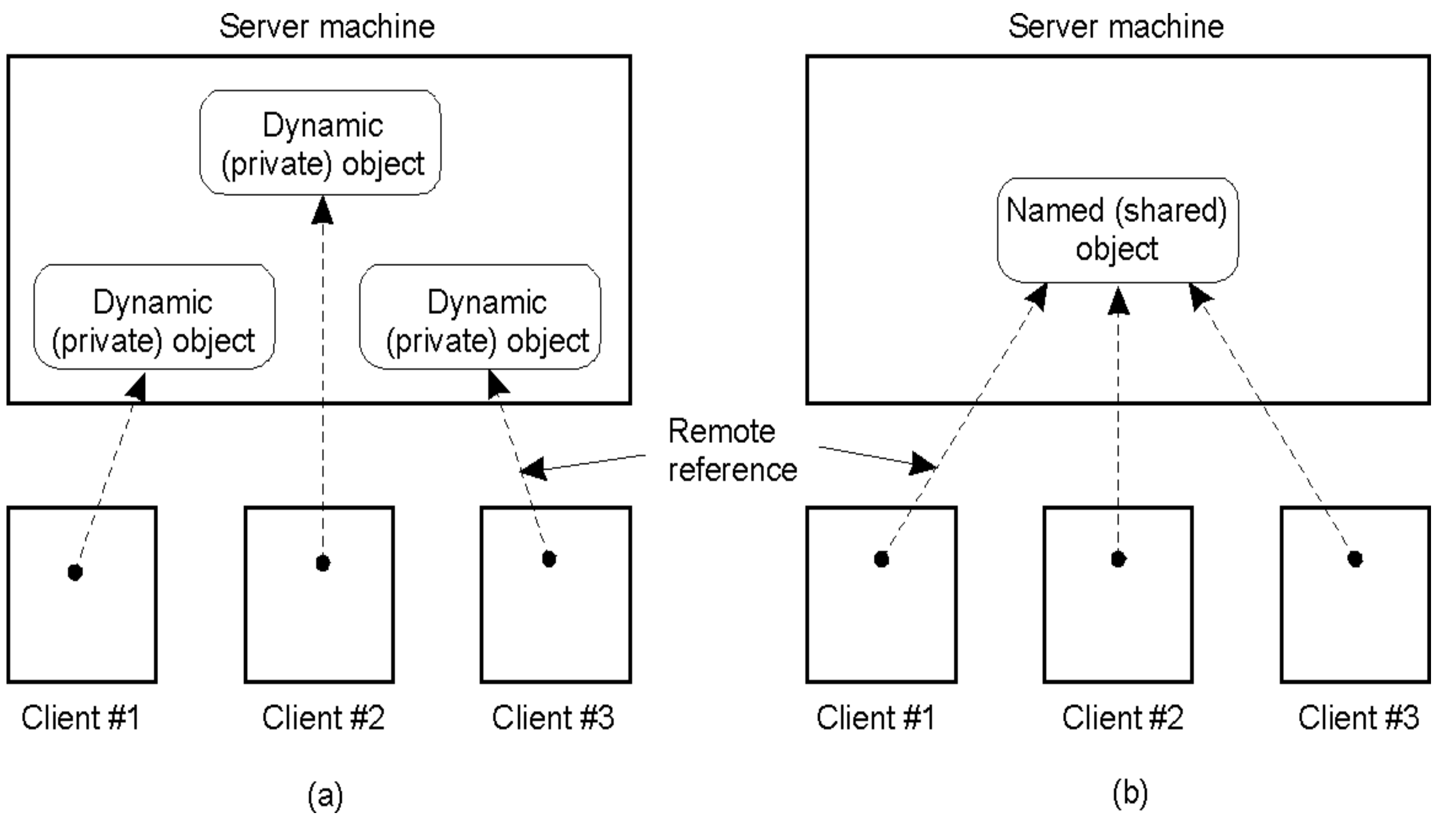
- **Proxy: client stub**
  - Maintains server ID, endpoint, object ID
  - Sets up and tears down connection with the server
  - [Java:] does serialization of local object parameters
  - In practice, can be downloaded/constructed on the fly (why can’t this be done for RPCs in general?)
- **Skeleton: server stub**
  - Does deserialization and passes parameters to server and sends result to proxy

Parameter Passing

- Less restrictive than RPCs.
  - Supports system-wide object references
  - [Java] pass local objects by value, pass remote objects by reference



Synchronization



a) Distributed dynamic objects
b) Distributed /shared named objects

Java RMI

- Allows a Java program on one machine to invoke a method on a remote object.
- **Server**
  - Defines interface and implements interface methods
  - Server program
    - Creates server object and registers object with “remote object” registry
- **Client**
  - Looks up server in remote object registry
  - Uses normal method call syntax for remote methods
- Java tools
  - **Rmiregistry**: server-side name server
  - **Rmic**: uses server interface to create client and server stubs

Java RMI

- RMI and RPC differs in two ways:
  1. RMI is object based: It supports invocation of methods on remote objects rather than procedures.
  2. The parameters to remote procedures are ordinary data structures in RPC; with RMI it is possible to pass objects as parameters to remote methods.

- If the parameters are local (non remote) objects, they are passed by copy using object serialization.
  - Object serialization allows the state of an object to be written to a byte stream.

Java RMI

The interface

```
package com.myapp.rmiinterface;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RMIInterface extends Remote
{
     public String helloTo(String name) throws
            RemoteException;
}
```

Java RMI

The Server

```
package com.mkyong.rmiserver;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject; import
com.myapp.rmiinterface.RMIInterface;

public class ServerOperation extends
UnicastRemoteObject implements RMIInterface
{
   private static final long serialVersionUID = 1L;

  protected ServerOperation() throws RemoteException
  {
     super();
  }
```

Java RMI

The Server (Procedure)

```
@Override public String helloTo(String name) throws
RemoteException
{

   System.err.println(name+ "is trying to contact!");

   return "Server says hello to " + name;
}
}
```

Java RMI

The Server (Main)

```
public static void main(String[] args)
{
   try
   {
         Naming.rebind("//localhost/MyServer", new
                                                 ServerOperation());

      System.err.println("Server ready");
   } catch (Exception e)
   {
      System.err.println("Server exception: " +
       e.toString()); e.printStackTrace();
   }
}
}
```

Java RMI

The Client

```
package com.myapp.rmiclient;
import  com.myapp.rmiinterface.RMIInterface;;
…
public class ClientOperation
{
   private static RMIInterface look_up;
   public static void main(String[] args) throws
MalformedURLException, RemoteException,
NotBoundException
{
     look_up = (RMIInterface)
     Naming.lookup("//localhost/MyServer");
     String response = look_up.helloTo(“John”);
     System.out.println(response);
}
}
```

Java RMI and Synchronization

- Java supports Monitors: synchronized objects
  - Serializes accesses to objects
  - How does this work for remote objects?
- Options: block at the client or the server
- Block at server
  - Can synchronize across multiple proxies
  - Problem: what if the client crashes while blocked?
- Block at proxy
  - Need to synchronize clients at different machines
  - Explicit distributed locking necessary
- Java uses proxies for blocking
  - No protection for simultaneous access from different clients
  - Applications need to implement distributed locking

CORBA

CORBA (Common Object Request Broker Architecture) standard

- Enables an object written in one programming language, running on one platform to interact with objects across the network that are written in other programming languages and running on other platforms.
  - For example, a client object written in C++ and running under Windows
  - can communicate with an object on a remote machine written in Java running under UNIX.

OMG

- The CORBA specification was developed by the Object Management Group (OMG).
- The OMG is an international, not-for-profit group consisting of approximately 800 companies and organizations defining standards for distributed object computing
- They are also behind other key object oriented standards such as UML (Unified Modeling Language).

CORBA

- The users of CORBA were diverse - including The Weather Channel, GNOME, US Army, CNN, and Charles Schwab.
- Several implementations of the CORBA standard exist.
  - Among the most widely used are IBM's SOM (a.k.a. SOMobjects) and DSOM architectures.
  - The Enterprise Edition of IBM's WebSphere (a software platform to help build and deploy high performance web sites) integrated CORBA (as well as Enterprise Java Beans) to build highly transactional, high-volume e-business applications

The Primary Elements

- IDL
  - Interface Definition Language
- Client / Server CORBA Objects
  - Abstract objects based upon a concrete implementation
- ORBs
  - Object Request Brokers
- GIOP / IIOP
  - General and Internet Inter-Object Protocols

Interface Definition Language

- Defines public interface for any CORBA server.
- C++ like syntax
- Client and Server implemented based on compilation of the same IDL (usually)
- OMG has defined mappings for:
  - C, C++, Java, COBOL, Smalltalk, ADA, Lisp, Python, and IDLscript

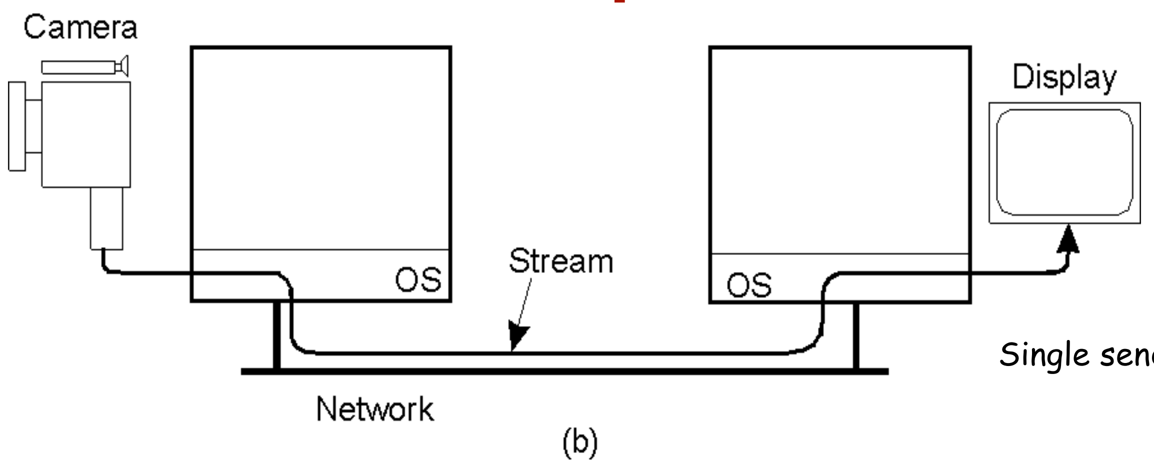
CORBA vs. DCOM vs. Java RMI

- DCOM
  - DCOM supports an object-oriented model, but differs substantially from classical OO models. DCOM object provides services through one or more distinct interfaces.
  - DCOM lacks polymorphism
  - CORBA can be deployed far more widely than DCOM and runs in many OS environment, while DCOM is running almost exclusively in the Windows environment.
- Java/RMI
  - JAVA/RMI systems fall short of seamless integration because of their interoperability requirements with other languages.
  - JAVA/RMI system assumes the homogeneous environment of the JVM, which can only take advantage of Java Object Model.

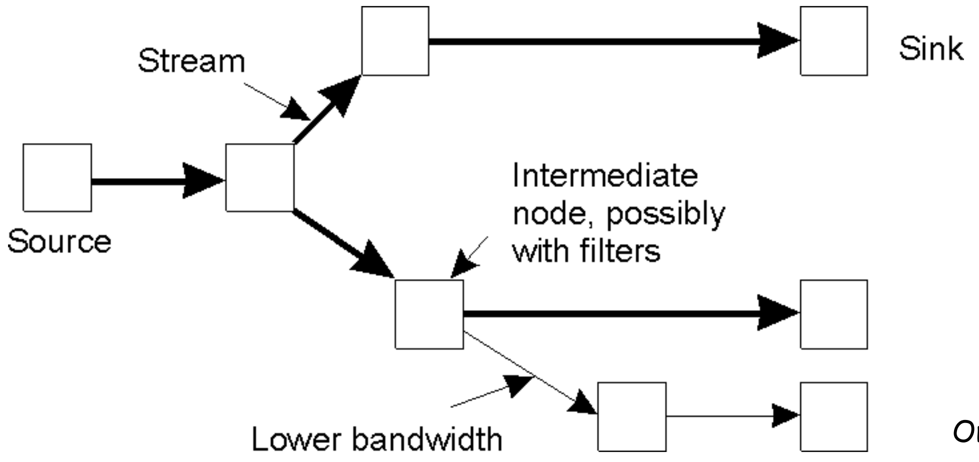
Stream Oriented Communication

- Message-oriented communication: request-response
  - When communication occurs and speed do not affect correctness
- Timing is crucial in certain forms of communication
  - Examples: audio and video (“continuous media”)
  - 30 frames/s video => receive and display a frame every 33ms
- Characteristics
  - Isochronous communication
    - Data transfers have a **maximum bound** on end-end delay and jitter
  - Push mode: no explicit requests for individual data units beyond the first “play” request

Examples



Single sender and receiver



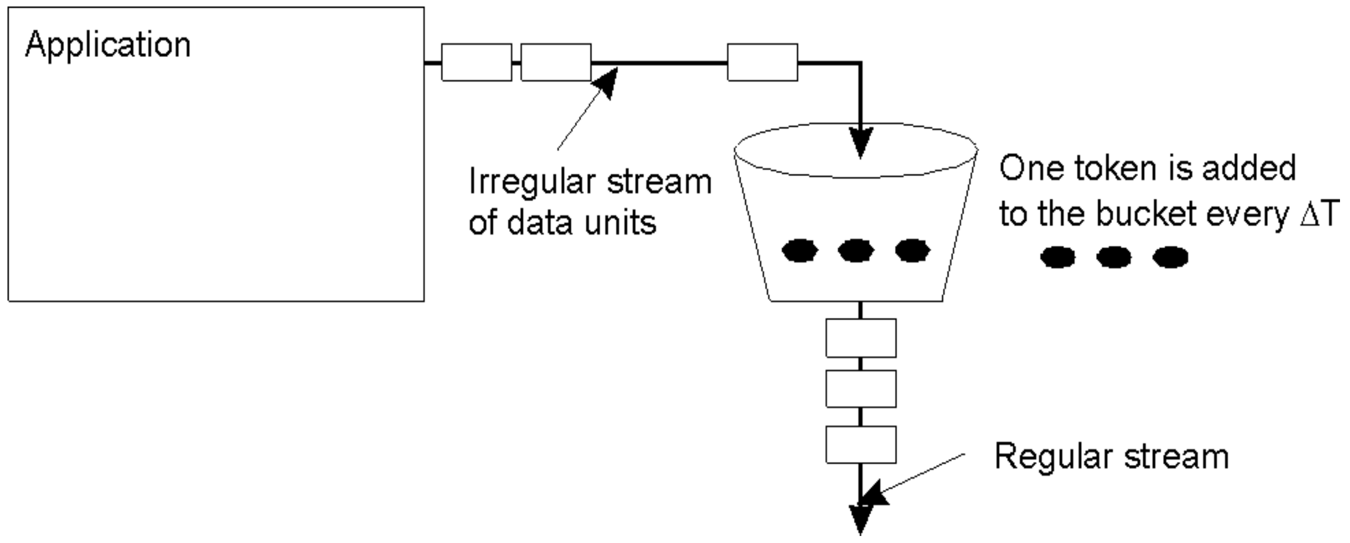
One sender Multiple receivers

Streams and Quality of Service

<u>QoS</u> is a way to encode the requirements of audio and video stream. (<u>many times QoS is instructive</u>).

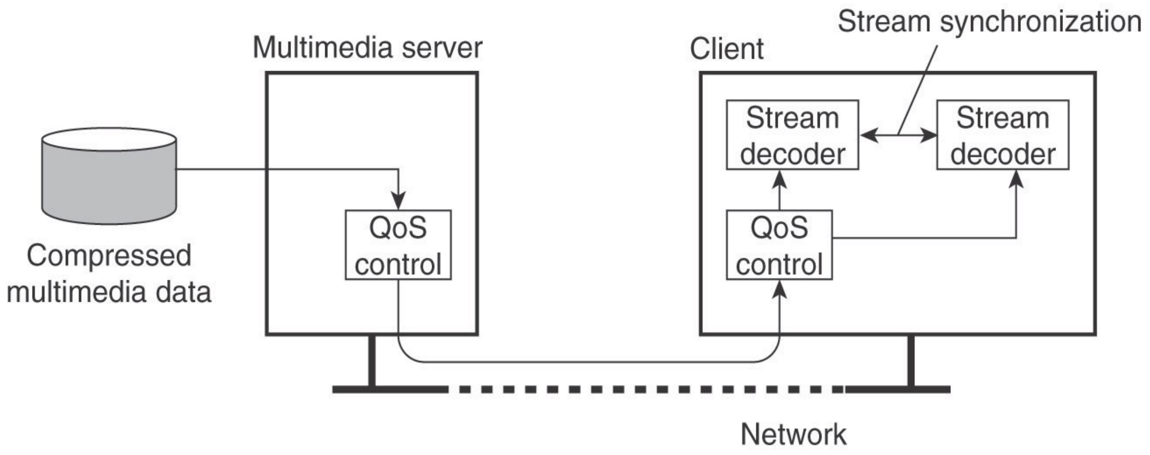
- Properties for Quality of Service:
- The required bit rate at which data should be transported.
- The maximum delay until a session has been set up
- The maximum end-to-end delay .
- The maximum delay variance, or jitter.
- The maximum round-trip delay.

Specifying QoS: Token bucket


Application
Irregular stream of data units
One token is added to the bucket every ΔT
Regular stream

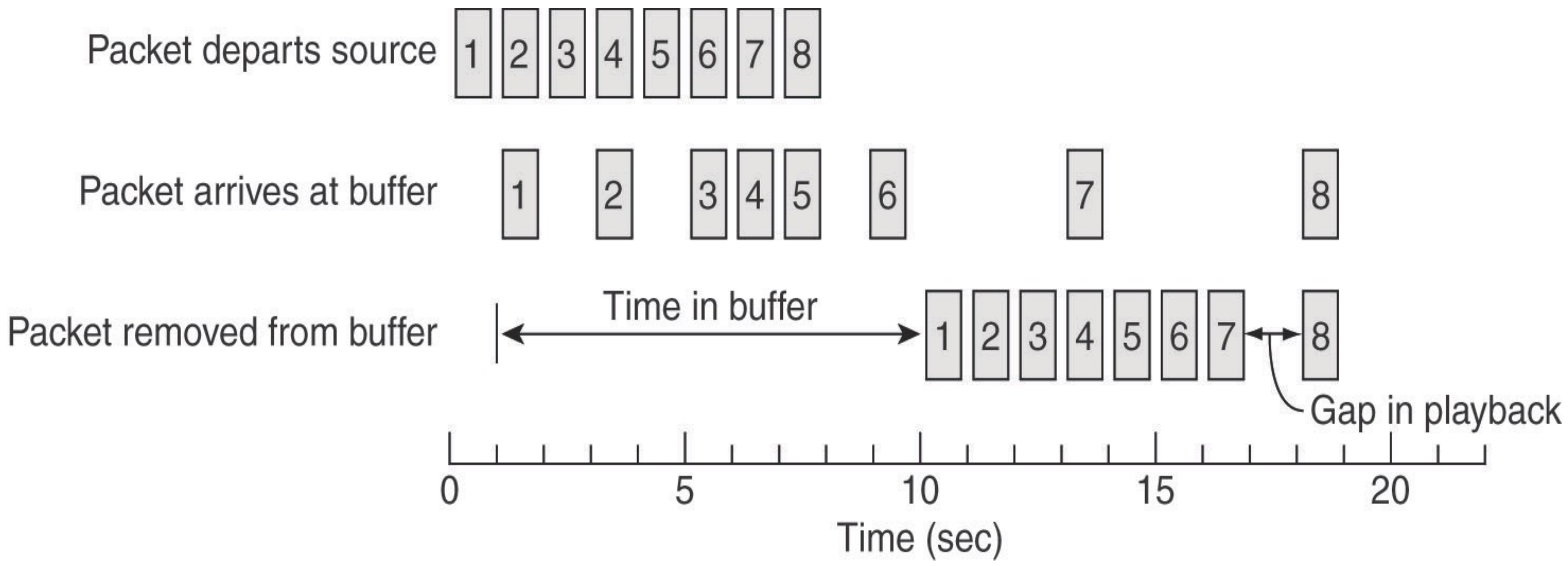
- The principle of a token bucket algorithm
  - Parameters (rate r, burst b)
  - Rate is the average rate, burst is the maximum number of packets that can arrive simultaneously

Enforcing QoS

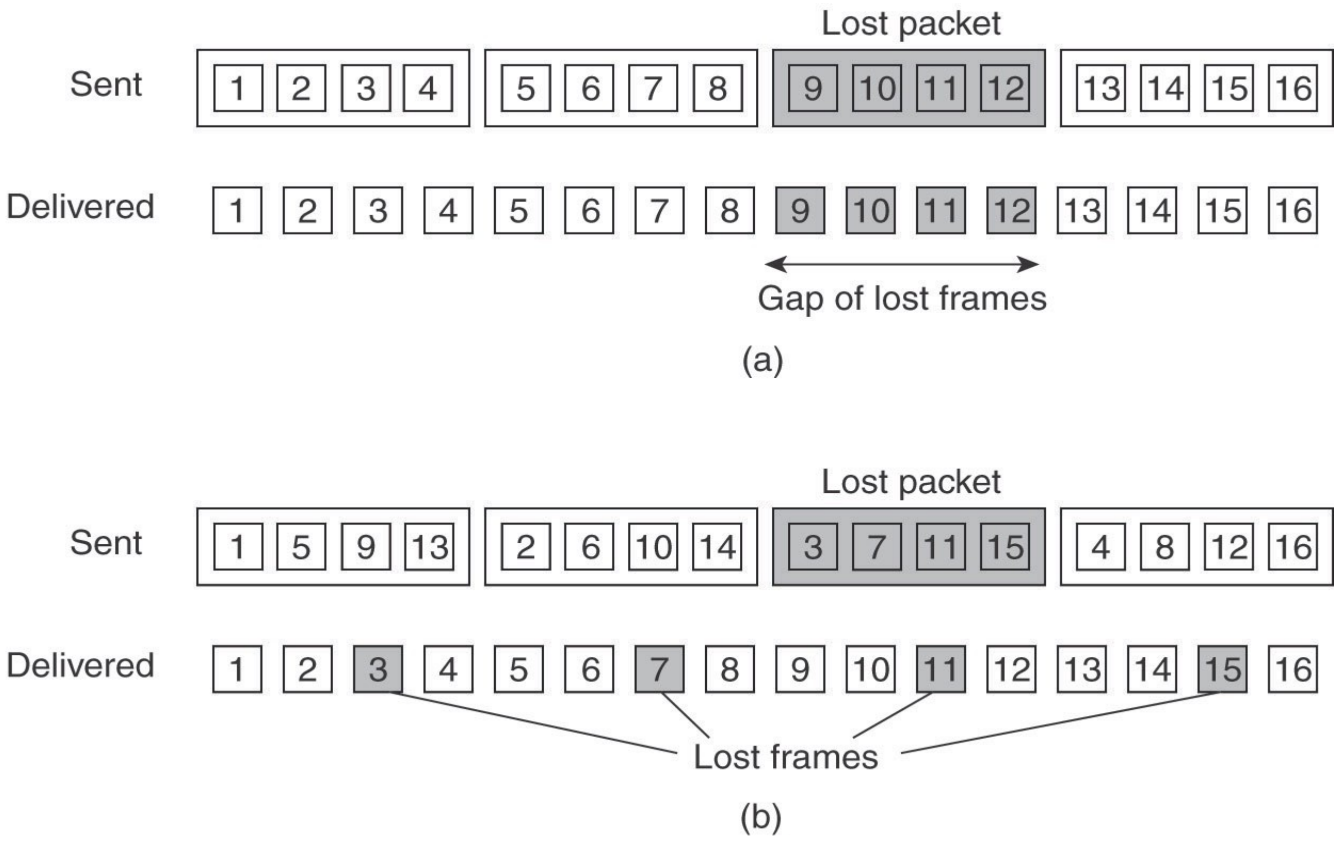


- Enforce at end-points (e.g., token bucket)
  - No network support needed
- Mark packets and use router support
  - Differentiated services: expedited & assured forwarding
- Use buffers at receiver to mask jitter
- Packet losses
  - Handle using forward error correction
  - Use interleaving to reduce impact

Enforcing QoS (1)



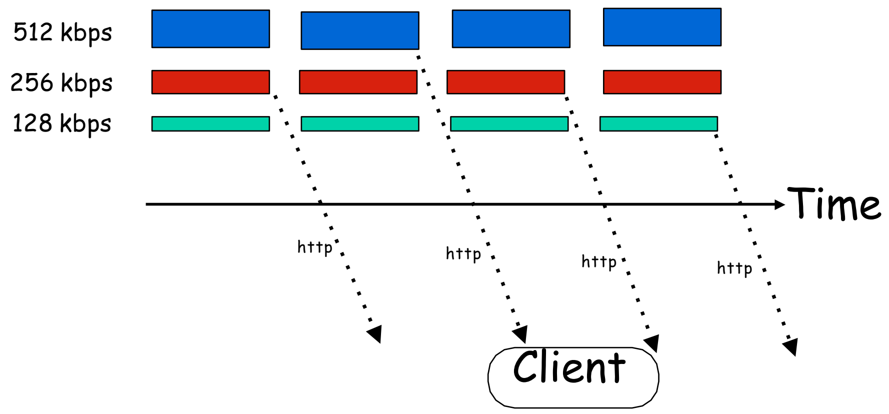
Enforcing QoS (2)



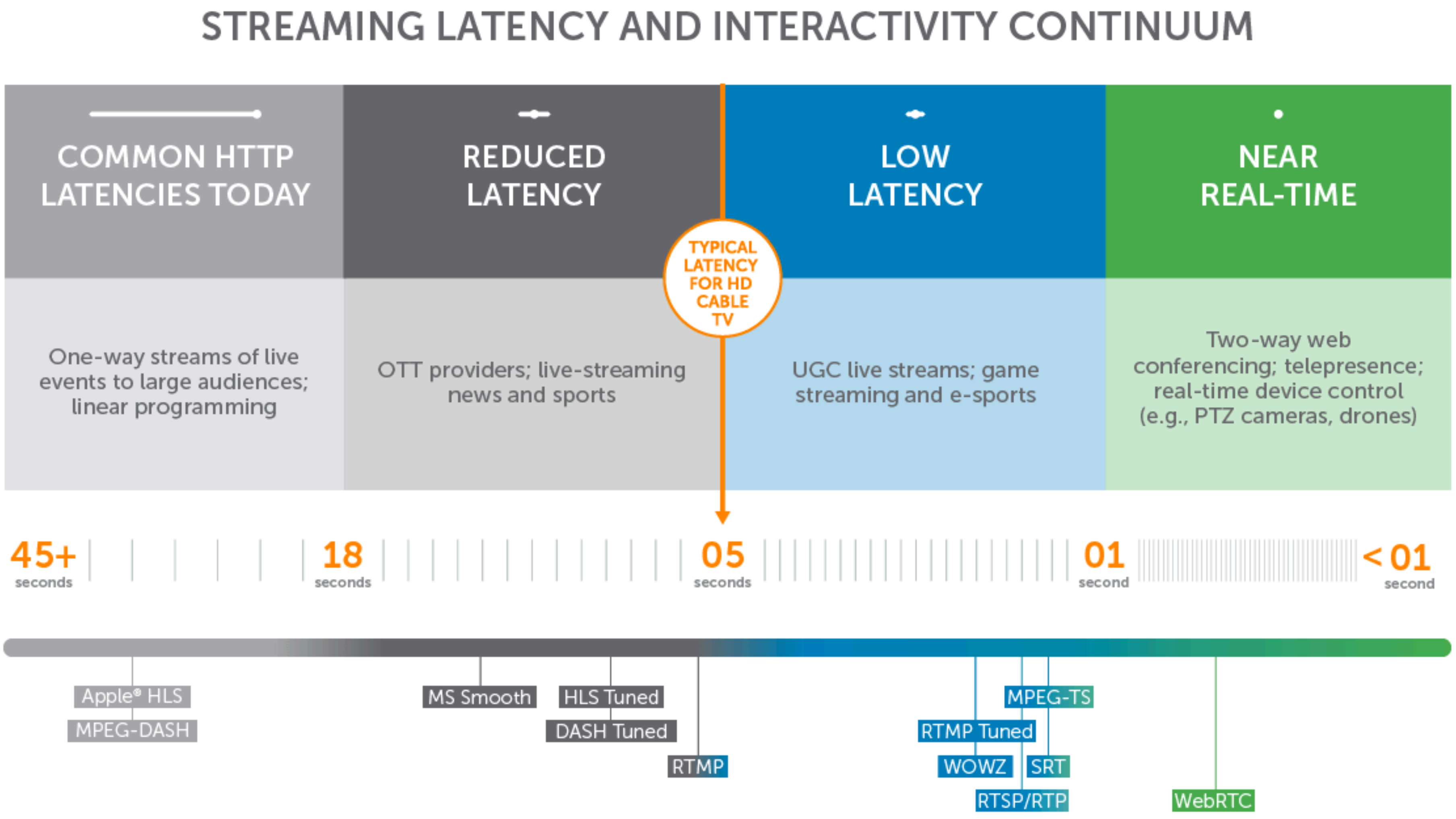
Can also use forward error correction (FEC)

HTTP Streaming

- UDP is inherently better suited for streaming
  - Adaptive streaming, specialized streaming protocols
- Yet, almost all streaming occurs over HTTP (and TCP)
  - Universal availability of HTTP, no special protocol needed
- Direct Adaptive Streaming over HTTP (DASH)
  - Intelligence is placed at the client



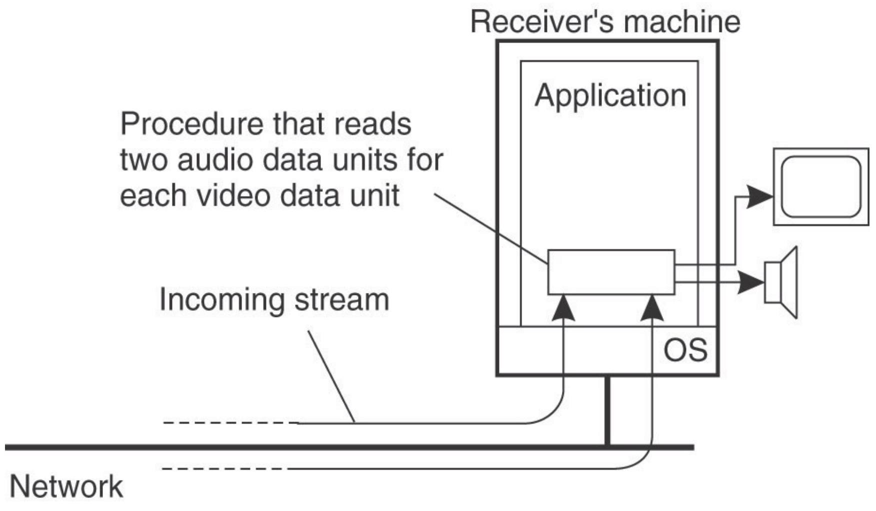
Streaming Protocols


STREAMING LATENCY AND INTERACTIVITY CONTINUUM
COMMON HTTP LATENCIES TODAY
REDUCED LATENCY
LOW LATENCY
NEAR REAL-TIME
TYPICAL LATENCY FOR HD CABLE TV
One-way streams of live events to large audiences; linear programming
OTT providers; live-streaming news and sports
UGC live streams; game streaming and e-sports
Two-way web conferencing; telepresence; real-time device control (e.g., PTZ cameras, drones)
45+ seconds
18 seconds
05 seconds
01 second
< 01 second
Apple® HLS
MPEG-DASH
MS Smooth
HLS Tuned
DASH Tuned
RTMP
MPEG-TS
RTMP Tuned
WOWZ
SRT
RTSP/RTP
WebRTC

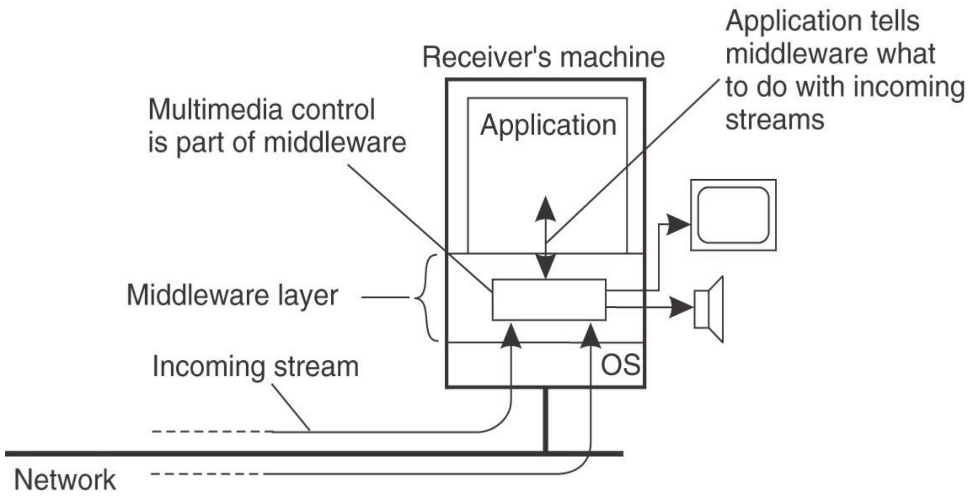
Stream synchronization

- Multiple streams:
  - Audio and video; layered video
- Need to sync prior to playback
  - Timestamp each stream and sync up data units prior to playback
- Sender or receiver?
- App does low-level sync
  - 30 fps: image every 33ms, lip-sync with audio
- Use middleware and specify playback rates

Synchronization Mechanism



Receiver's machine
Application
Procedure that reads two audio data units for each video data unit
Incoming stream
OS
Network



Receiver's machine
Application tells middleware what to do with incoming streams
Multimedia control is part of middleware
Application
Middleware layer
Incoming stream
OS
Network