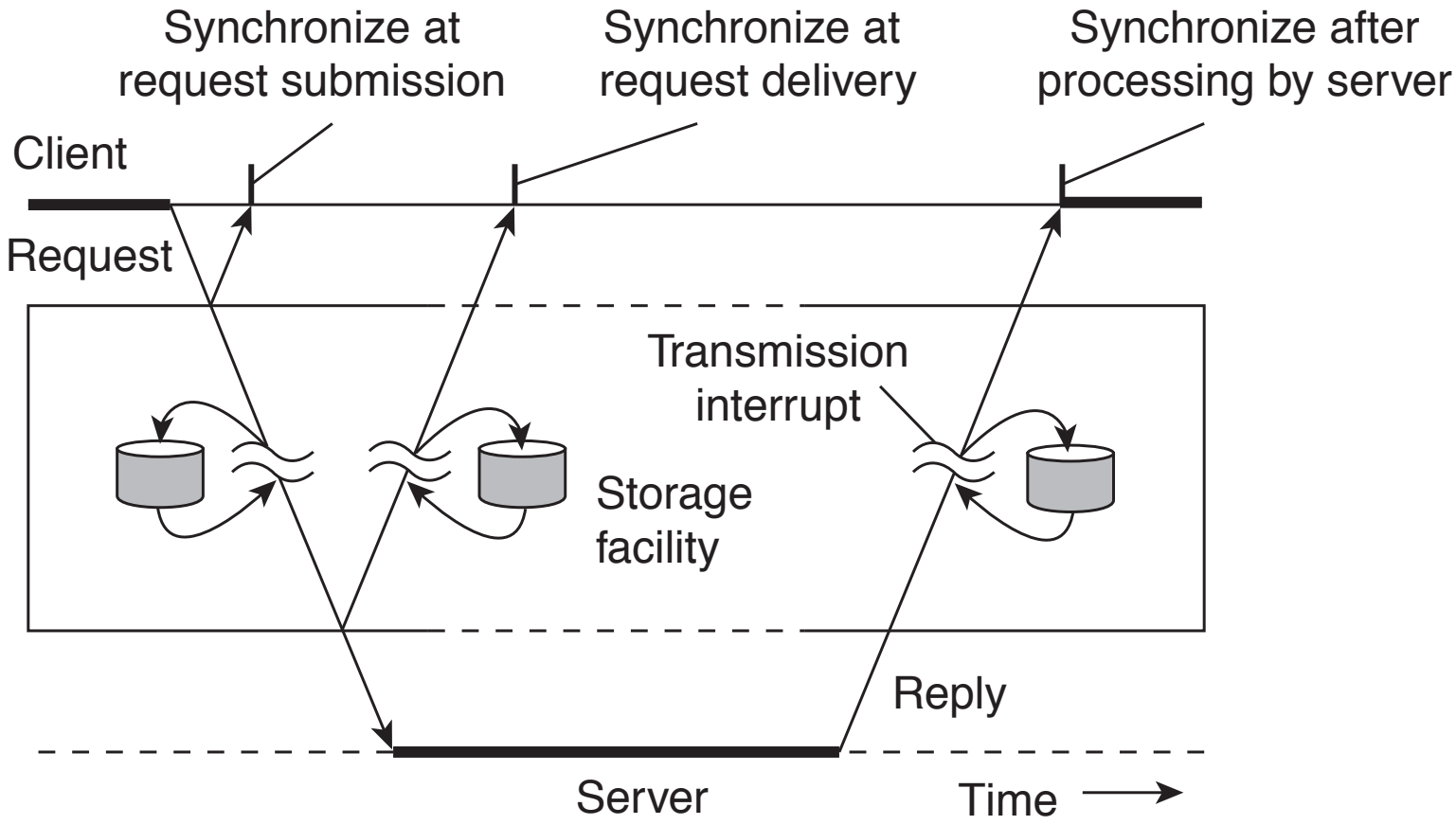
Remote Procedure Call

Yao Liu

Types of communication

- Asynchronous vs. synchronous communication
- Transient vs. persistent communication



Remote procedure call

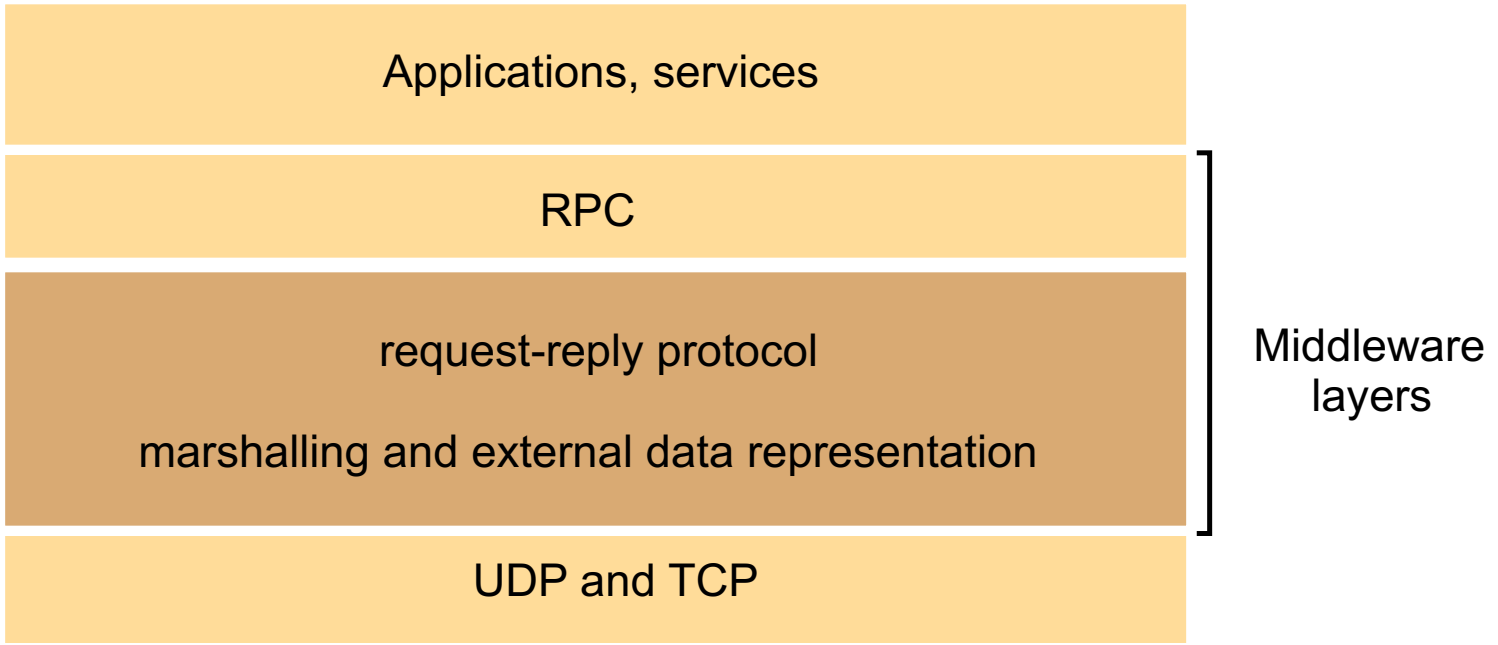
- RPC (by default) supports communication between two processes that are executing at the same time
  - transient communication
- Typically, the client is blocked until the RPC returns
  - synchronous communication

Remote procedure call

- Goal: to provide a procedural interface for distributed (i.e., remote) services

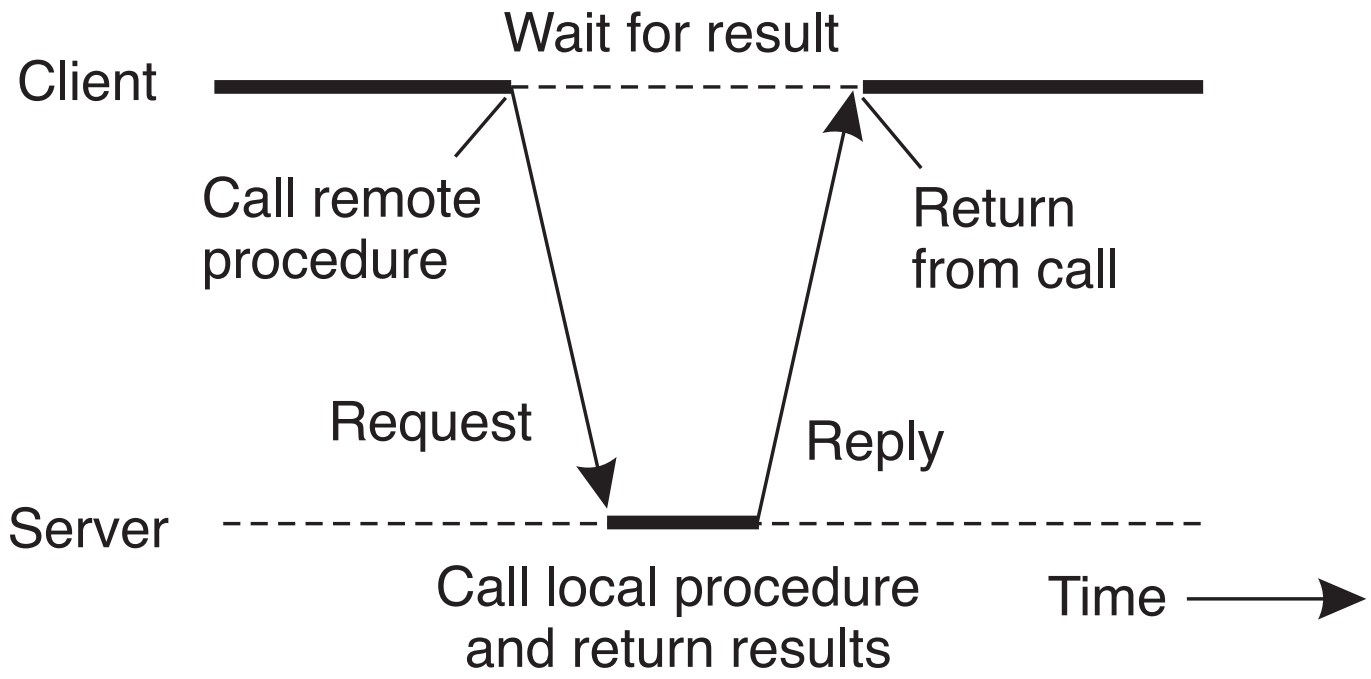
- To make distributed nature of service transparent to the programmer

Middleware layers

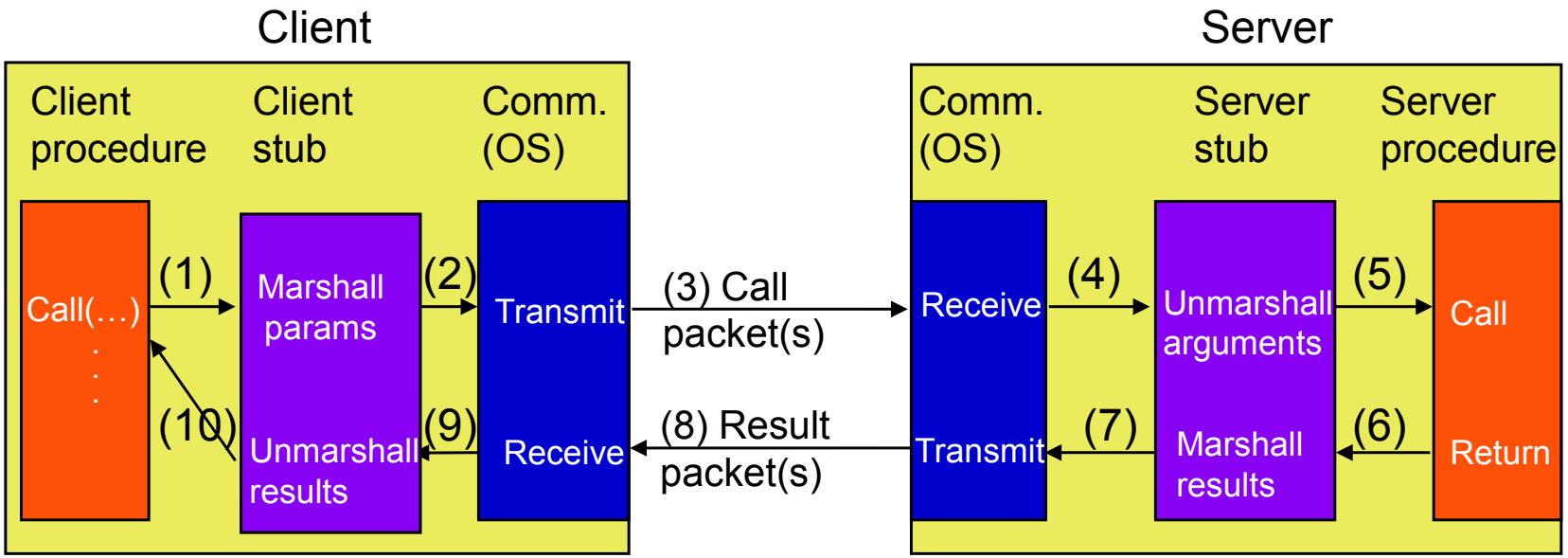


Remote procedure call

- Principle of RPC between a client and server program.

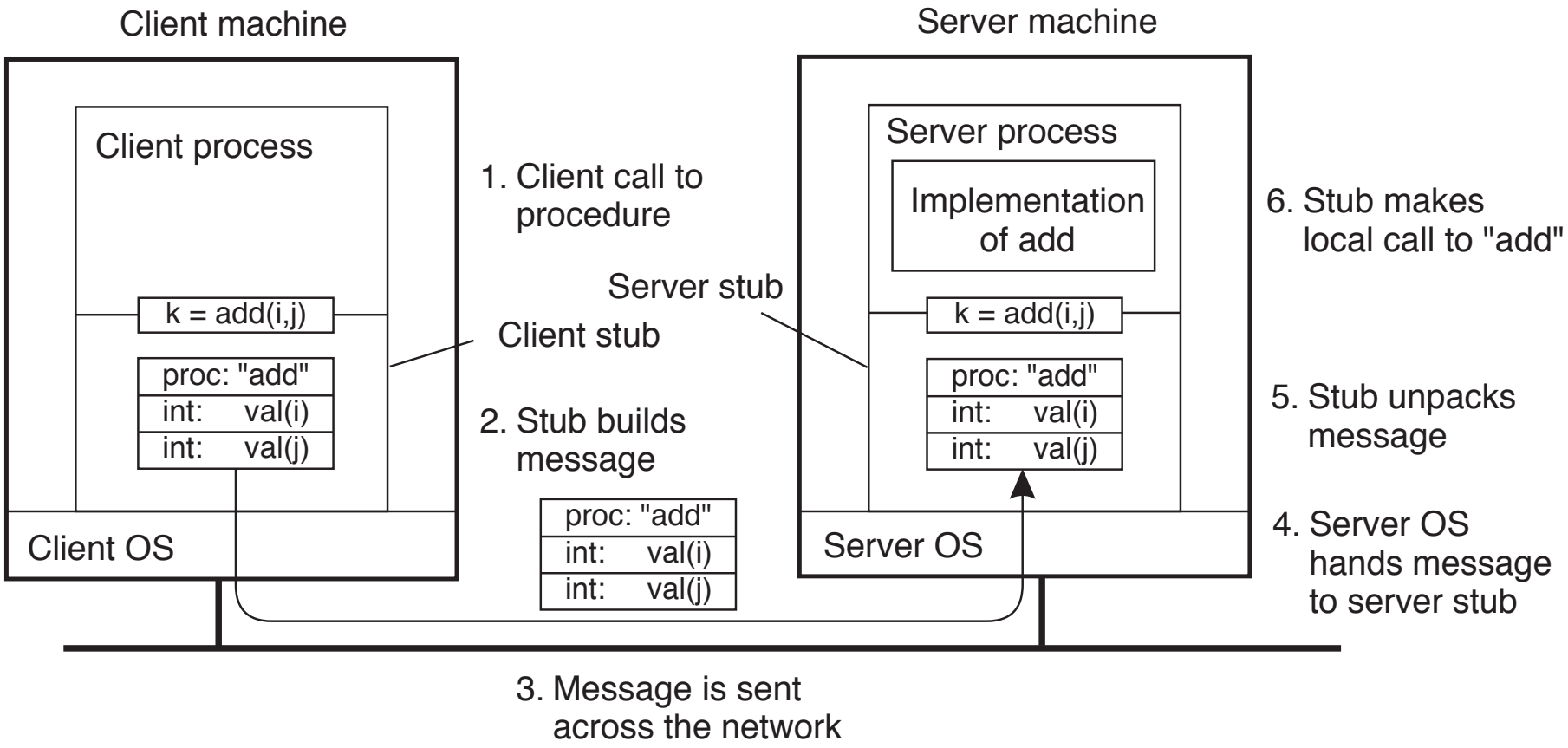


Steps of a remote procedure call



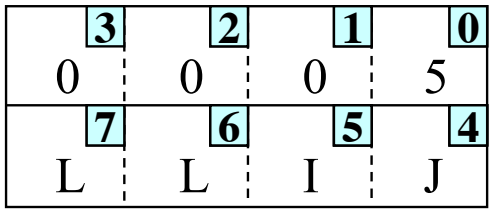
1) Client procedure calls client stub in normal way
2) Client stub builds message, calls local OS
3) Client's OS sends message to remote OS
4) Remote OS gives message to server stub
5) Server stub unpacks parameters, calls server
6) Server does work, returns result to the stub
7) Server stub packs it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

Passing value parameters (1)

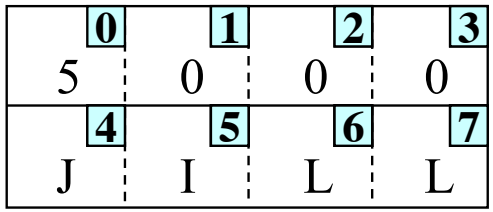


Passing value parameters (2)

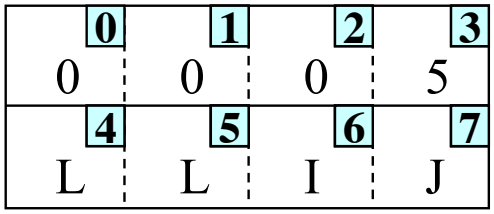
- Data representation: not a problem when processes are on the same machine
- Different machines have different representations
- e.g., different computers may store bytes of an integer in different order in memory (e.g., little endian, big endian), called “host byte order”



Original message on Intel 486



Message after receipt on SPARC



Message after being inverted

The little numbers in boxes indicate the address of each byte

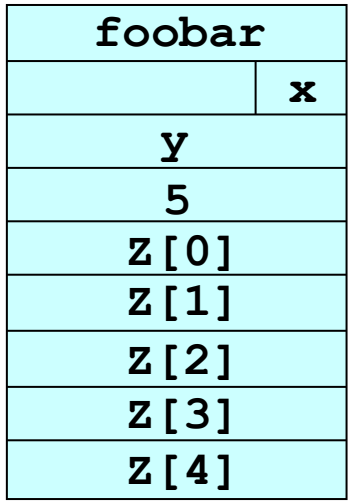
Passing reference parameters

- Call-by-reference not possible: the client and server don’t share an address space. That is, addresses referenced by the server correspond to data residing in the client's address space.
- One approach is to simulate call-by-reference using copy-restore. In copy-restore, call-by-reference parameters are handled by sending a copy of the referenced data structure to the server, and on return replacing the client's copy with that modified by the server.
- However, copy-restore doesn’t work in all cases. For instance, if the same argument is passed twice, two copies will be made, and references through one parameter only changes one of the copies.

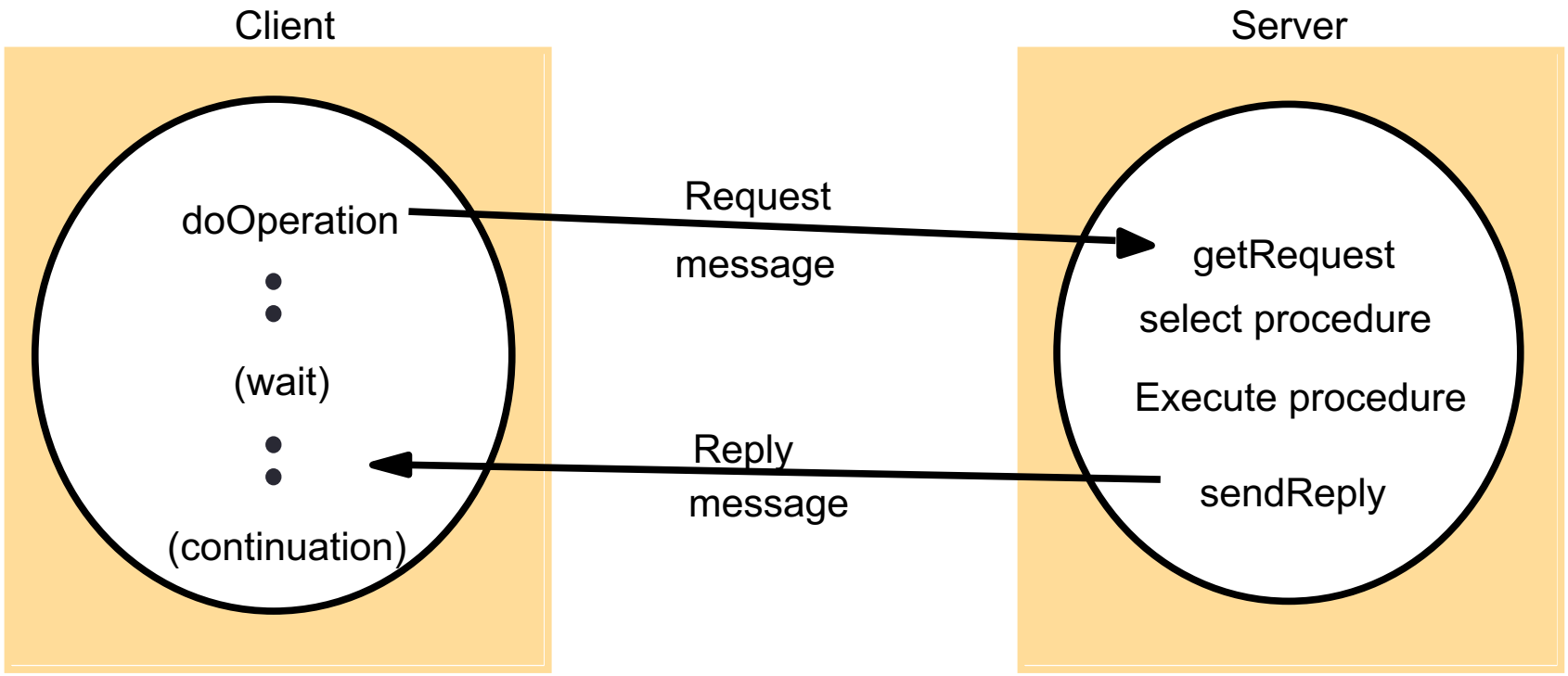
Parameter specification and stub generation

- Once a standard has been agreed upon for representing each of the basic data types, given a parameter list and a message, it is possible to deduce which bytes belong to which parameter, and thus to solve the problem
- Given these rules, the requesting process knows that it must use this format, and the receiving process knows that incoming message will have this format
- Having the type information of parameters makes it possible to make necessary conversions

```
foobar (x,y,z)
   char x;
   float y;
   int z[5];
{
…
}
```



Request-reply communication



Style of request-reply protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)

sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

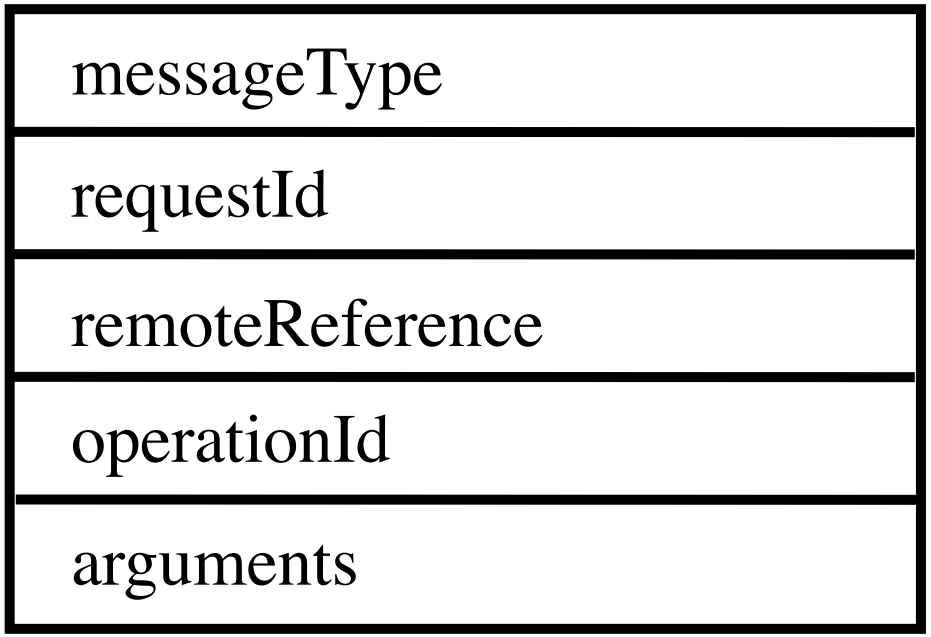
public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

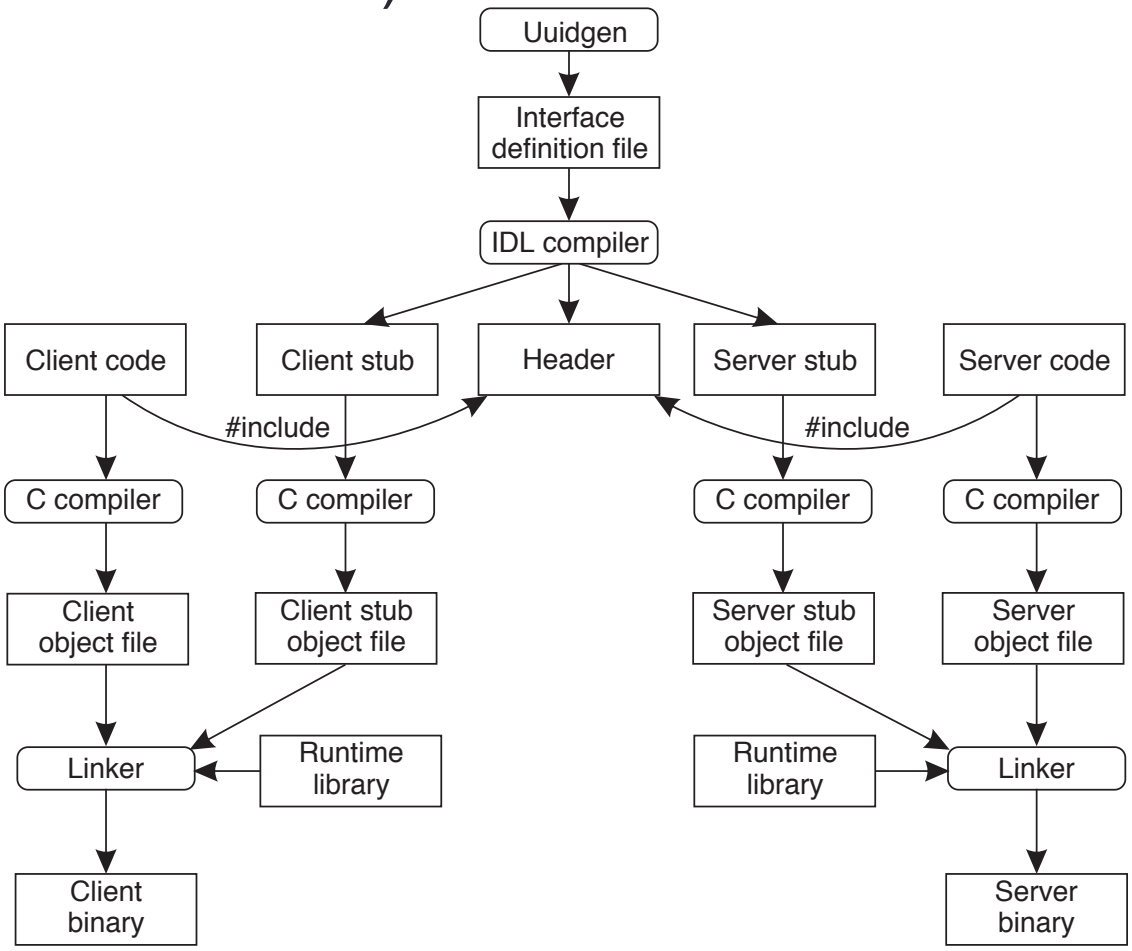
sends the reply message reply to the client at its Internet address and port.

Request-reply message structure



Writing a client and a server

- The steps in writing a client and a server in DCE RPC (SUN RPC is similar)



Writing a client and a server (2)

- Three files output by the IDL compiler:

- A header file (e.g., interface.h, in C terms).
- The client stub.
- The server stub.

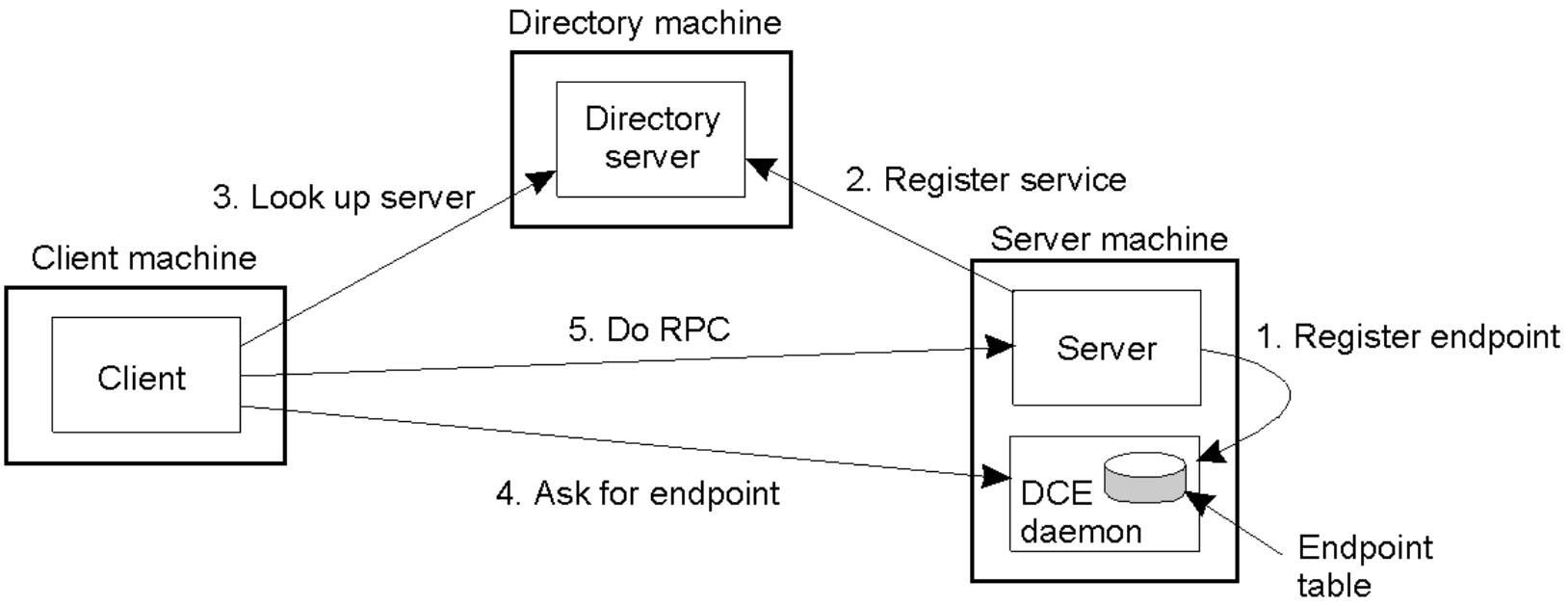
Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;

struct Data {
  int length;
  char buffer[MAX];
};

struct writeargs {
  FileIdentifier f;
  FilePointer position;
  Data data;
};
```

```
struct readargs {
  FileIdentifier f;
  FilePointer position;
  Length length;
};

program FILEREADWRITE {
  version VERSION {
    void WRITE(writeargs)=1;
    Data READ(readargs)=2;
  }=2;
}=9999;
```

Binding a client to a server (1)

- Client-to-server binding in DCE.



Binding a client to a server (2)

- Registration of a server makes it possible for a client to locate the server and bind to it.

- Server location is done in two steps:
  1. Locate the server's machine.
  2. Locate the server on that machine.

RPC in presence of failures

- Five different classes of failures can occur in RPC systems
  - The client is unable to locate the server
  - The request message from the client to the server is lost
  - The reply message from the server to the client is lost
  - The server crashes after receiving a request
  - The client crashes after sending a request

Client cannot locate the server

- Examples:
  - Server might be down
  - Server might have moved
  - Server evolves (new version of the interface installed and new stubs generated) while the client is compiled with an older version of the client stub
- Possible solutions:
  - Use special code, such as “-1”, as the return value of the procedure to indicate failure. In Unix, add a new error type and assign the corresponding value to the global variable errno.
    - -1 can be a legal value to be returned, e.g., sum(7, -8)
  - Have the error raise an exception or a signal.
    - Writing an exception/signal handler destroys the transparency…

Lost request message

- Client starts a timer when sending the message:
  - If timer expires before reply or ACK comes back: client retransmits
  - If message truly lost: server will not differentiate between original and retransmission → everything will work fine
  - If many requests are lost: client gives up and falsely concludes that the server is down → we are back to "Cannot locate server"

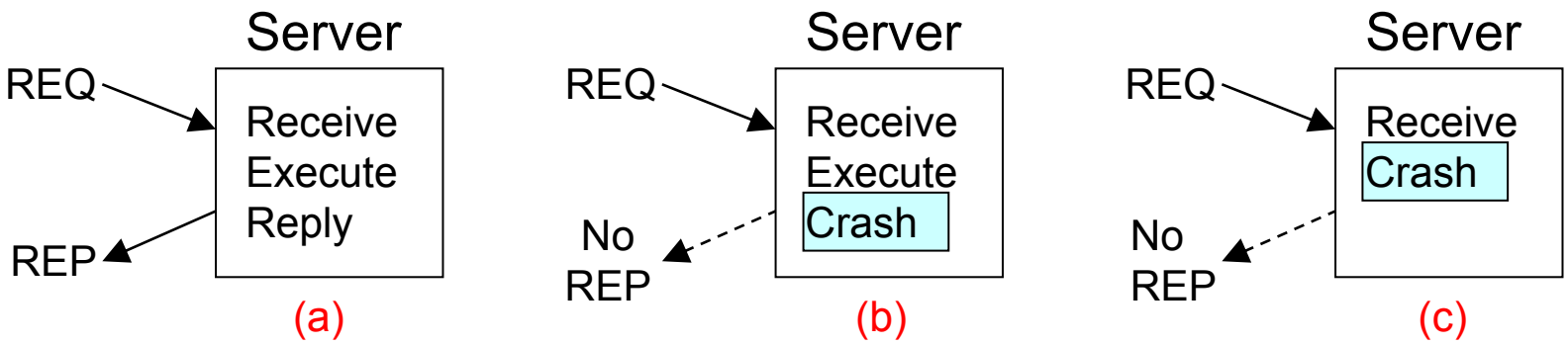
Lost reply message

- Client starts a timer when sending the message:
  - If timer expires before reply comes back: retransmit the request
    - Problem: not sure why no reply (reply/request lost or server slow) ?
  - If server is just slow: the procedure will be executed several times
    - Problem: what if the request is not idempotent, e.g., money transfer
- Server choices:
  - Re-execute procedure → service should be idempotent so that it can be repeated safely
  - Filter duplicates → server should hold on to results until acknowledged
    - Client assigns sequence numbers to requests to allow server to differentiate retransmissions from original

Invocation semantics

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Server crashes



- Problem: client cannot differentiate between (b) and (c)
  - Wait until the server reboots and try the operation again. Guarantees that RPC has been executed **at least one** time (at-least-once semantics).
  - Give up immediately and report back failure. Guarantees that RPC has been carried out **at most one** time (at-most-once semantics).
  - Client gets no help about what happened. Guarantees nothing. Easy to implement.

Client crashes

- Client sends a request and crashes before the server replies: a computation is active and no parent is waiting for result (orphan)
  - Orphans waste CPU cycles and can lock files or tie up valuable resources
  - Orphans can cause confusion (client reboots and does RPC again, but the reply from the orphan comes back immediately afterwards)
- Possible solutions
  - Extermination:
    - Before a client stub sends an RPC, it makes a log entry (in safe storage) telling what it is about to do. After a reboot, the log is checked and the orphan explicitly killed off.
    - Expense of writing a disk record for every RPC; orphans may do RPCs, thus creating grand-orphans impossible to locate; impossibility to kill orphans if the network is partitioned due to failure.

Client crashes (2)

- Possible solutions (cont'd)
  - Reincarnation:
    - Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring the start of a new epoch. When broadcast comes, all remote computations are killed. Solve problem without the need to write disk records
    - If network is partitioned, some orphans may survive. But, when they report back, they are easily detected given their obsolete epoch number
  - Expiration:
    - Each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum.
    - Choosing a reasonable value of T in the face of RPCs with wildly differing requirements is difficult

RPC recap

- Logical extension of the procedure call interface
  - Easy for the programmer to understand and use
- Requires an Interface Definition Language (IDL) to specify data types and procedure interfaces
  - Provides language independence
- IDL compiler generates client and server stubs
  - Stubs handle the marshalling and unmarshalling of arguments and builds the message
    - → Messages must be represented in a machine independent data representation
    - → Must serialize complex types
- Calling and called procedures reside in different address space
  - Cannot send pointers or system specific data (locks, file descriptors, pipes, sockets, etc.)
  - Parameter passing can therefore be very expensive

RPC recap (2)

- New failure models:
  - Cannot locate server (throw exception)
  - Lost request message (resend request)
  - Lost reply message
    - Retransmit request without incrementing the request sequence number
    - Requires that the server retain old replies for some set time period
  - Server crashes before sending reply
    - RPC resends request (at least once semantics)
    - Give up and report failure (at most once semantics)
      - Or server saves all replies on persistent storage and re-send previous replies
    - Client gets no help, semantics determined by the client
  - Client crashes

RPC example: Apache Thrift

- https://thrift.apache.org/about
- Developed at Facebook and open-sourced in 2007
- Used by many companies in their production services today:
  - Evernote, Facebook, last.fm, Pinterest, Quora, Uber, etc.
- Used in many other Apache projects today:
  - Hadoop, HBase, Storm, etc.

Apache Thrift

- Supports
  - C++, C#, Cocoa, D, Delphi, Erlang, Haskell, Java, OCaml, Perl, PHP, Python, Ruby, Smalltalk
- Different data encoding types
  - Binary, JSON
- IDL provides relatively simple datatypes
  - Primitives
  - Simple structures (structs, lists, sets, maps)

Apache Thrift

- Specifies its own IDL

```
enum Operation {
  ADD = 1,
  SUBTRACT = 2,
  MULTIPLY = 3,
  DIVIDE = 4
}
```

```
struct Work {
  1: i32 num1 = 0,
  2: i32 num2,
  3: Operation op,
  4: optional string comment,
}
```

```
service Calculator {
void ping(),
i32 add(1:i32 num1, 2:i32 num2),
i32 calculate(1:i32 logid, 2:Work w)
                throws (1:InvalidOperation ouch),
}
```

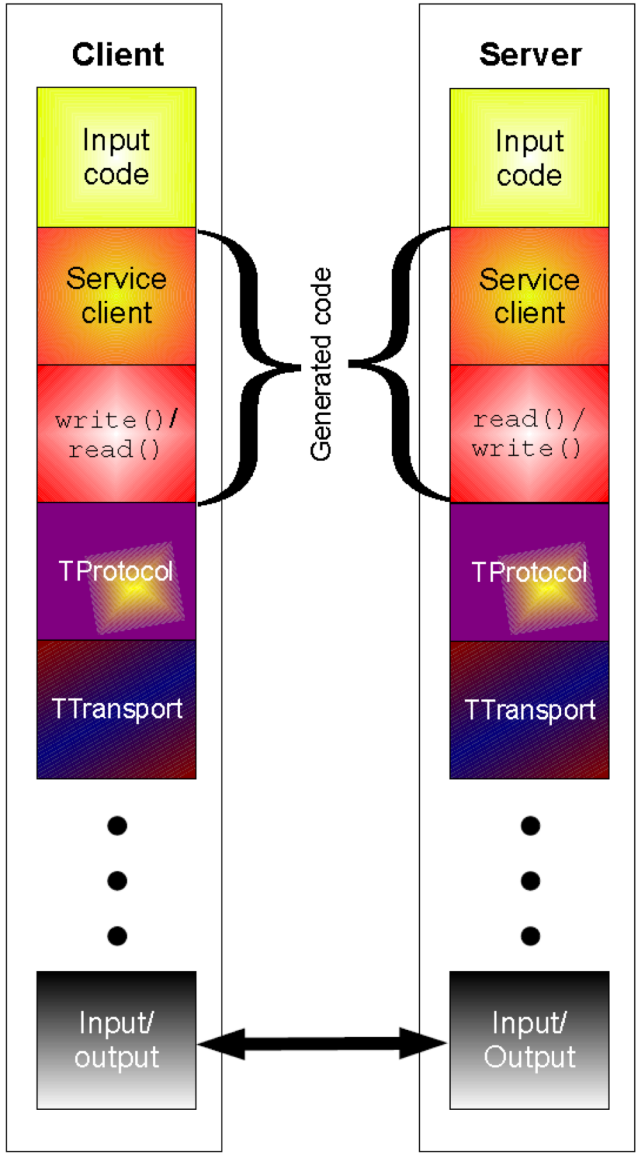
- IDL compiler generates client/server stubs
- Server has to implement the handler for the service

```
class CalculatorHandler:
    def __init__(self):
        #Your initialization goes here
    def ping(self):
        #Your implementation goes here
    def add(self, n1, n2):
        #Your implementation goes here
    def calculate(self, logid, work):
        #Your implementation goes here
```

```
public class CalculatorHandler implements Calculator.Iface {
    public CalculatorHandler() {
        // Your initialization goes here
    }
    public void ping() {
        // Your implementation goes here
    }
    public int add(int n1, int n2) {}
    public int calculate(int logid, Work work) throws InvalidOperation {}
}
```

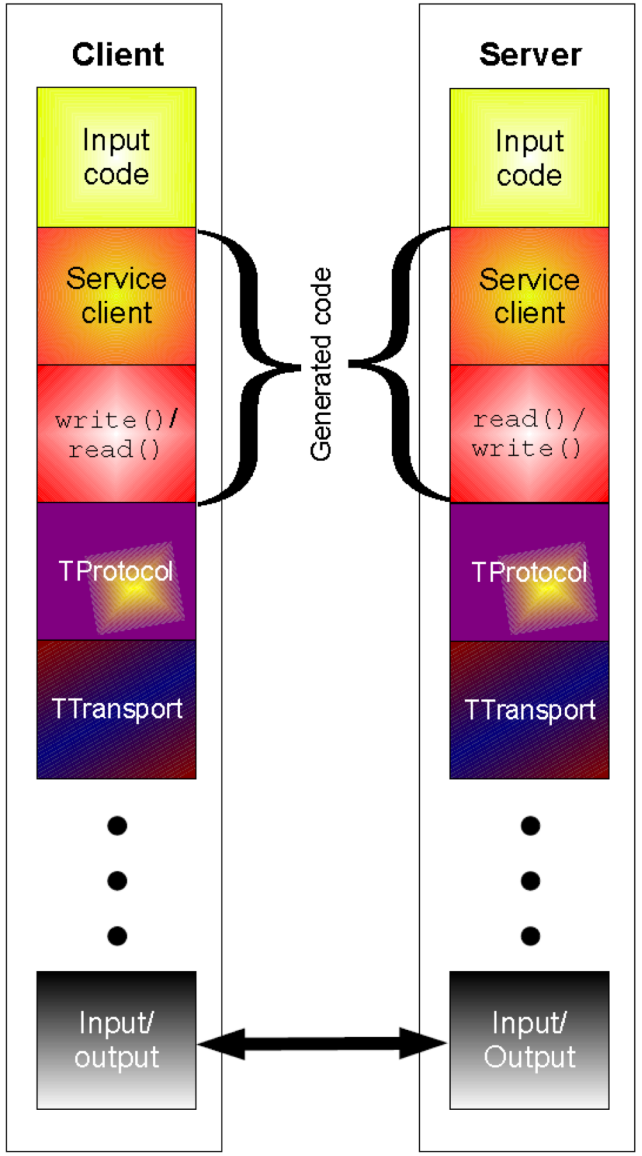
Apache Thrift

- Client and server consist of three main layers:
  - written code
  - generated code
  - the Apache Thrift libraries
- The written code allows to choose
  - marshalling mechanisms (TProtocol)
  - I/O mechanisms are used (TTransport)



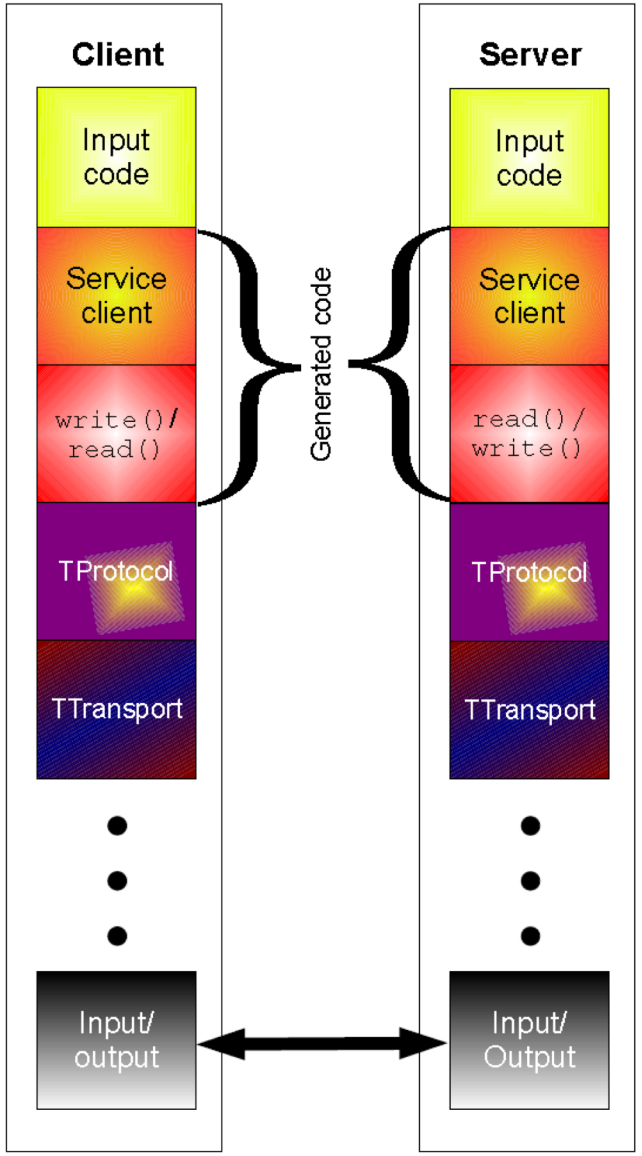
Apache Thrift

- Thrift Protocol defines:
  - What is transmitted?
  - i.e., how message is constructed, encoding, decoding, (marshalling)
- Thrift protocols (implementations of TProtocol interface)
  - TBinaryProtocol:
    - Numeric values are encoded binary instead of text
  - TCompactProtocol:
    - Efficient and dense encoding of data
  - TJSONProtocol:
    - User JSON for encoding of data
  - …



Apache Thrift

- Thrift Transport defines:
  - How is transmitted? In terms of I/O operations
- Thrift transports (implementations of TTransport interface)
  - TSocket
    - Using blocking I/O for transport
  - TNonblockingSocket
    - non-blocking I/O Operations
  - TFileTransport
    - Abstraction of an on-disk-file to a data stream.
  - …



Apache Thrift

- The real service procedures (e.g., CalculatorHandler) are encapsulated in a TProcessor object in the generated code.

```
handler = CalculatorHandler()
processor = Calculator.Processor(handler)
transport = TSocket.TServerSocket(port=9090)
tfactory = TTransport.TBufferedTransportFactory()
pfactory = TBinaryProtocol.TBinaryProtocolFactory()

server = TServer.TSimpleServer(processor, transport,
                               tfactory, pfactory)

server.serve()
```

Apache Thrift

- Go to https://thrift.apache.org/ for more information on the official Documentation and Tutorial.
- Also refer to a Whitepaper on the motivation and design choices made in Thrift, written by Facebook:
  - https://thrift.apache.org/static/files/thrift-20070401.pdf
- Thrift tutorial on September 24
  - https://github.com/vipulchaskar/thrift-lab

Google Protocol Buffer (protobuf)

- `protobuf` provides an efficient mechanism for serializing structured data
- i.e., only marshalling and un-marshalling
- No I/O mechanisms are provided

Protocol buffer definition example

```
message Person {
 required string name = 1;
 required int32 id = 2;
 optional string email = 3;
 enum PhoneType {
       MOBILE = 0;
       HOME = 1;
       WORK = 2;
 }
 message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
 }
 repeated PhoneNumber phones = 4;
}
```

protobuf in Python

```
person = Person()
person.id = 1234
person.name = “John Doe”
person.email = jdoe@binghamton.edu
phone = person.phone.add()
phone.number = “607-777-1000”
phone.type = Person.Work
```

protobuf in Java

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@binghamton.edu
");
```

Reading

- TBook
  - Section 4.1 on communication fundamentals
  - Section 4.2 on RPC
  - Section 8.3.2 on RPC in presence of failures