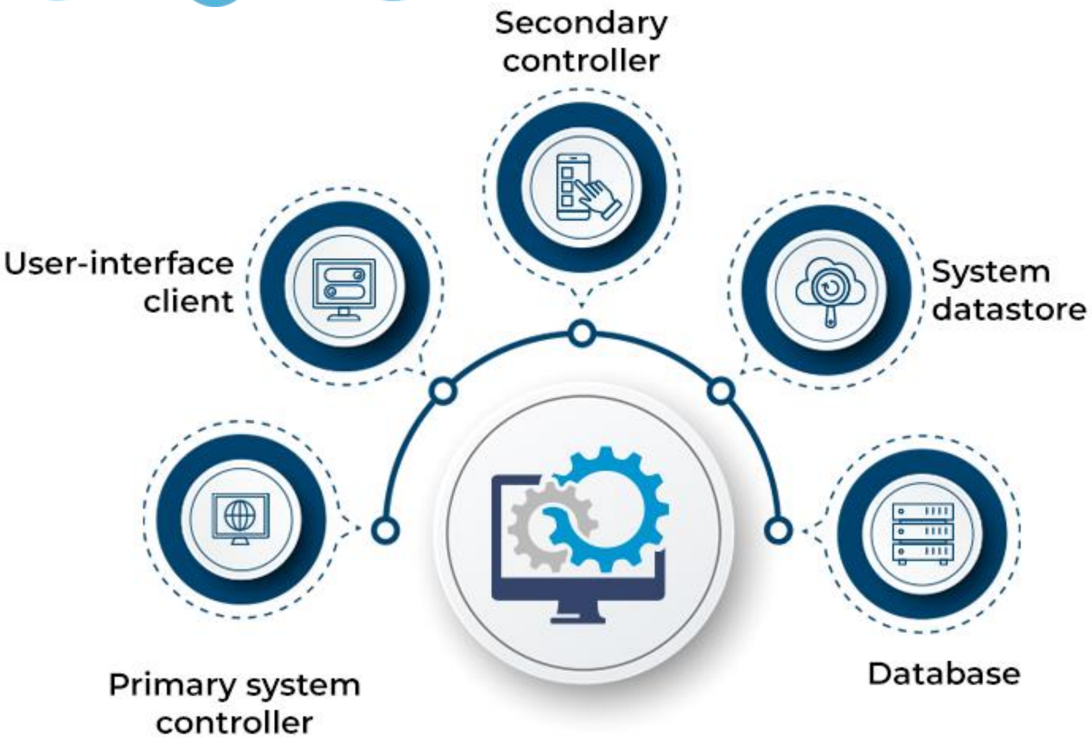
Sistem Terdistribusi

IF2222



04: RPC

2022-23 Ganjil

Teknik Informatika
Universitas Trunojoyo Madura

Sistem Terdistribusi 2022

1. Mengenal Sistem Terdistribusi
2. Review Jaringan Komputer (layer 2, 3, dan 4)
3. Arsitektur Sistem Terdistribusi
4. ***Remote Procedure Calls* (RPC)**
5. Layanan Penamaan
6. Sinkronisasi Data (2 pekan)
7. *Message Passing Interface* (MPI)
8. Contoh Arsitektur: Hadoop, Pregel, Blockchain
9. Teknik *Caching*
10. Teknik Replikasi Data (2 pekan)
11. Basis Data Terdistribusi
12. Toleransi Kegagalan

Capaian Pembelajaran

Kuliah ini bertujuan memberikan pemahaman mendalam dan pengalaman langsung tentang:

- Prinsip yang menjadi basis dari sistem terdistribusi
- Prinsip untuk optimalisasi sistem terdistribusi
- Model pemrograman sistem terdistribusi dan mesin analitika
- Bagaimana sistem terdistribusi modern memenuhi tuntutan aplikasi terdistribusi kontemporer

Today…

- Last Session:
  - Arsitektur Sistem Terdistribusi
- Today's Session:
  - Remote Procedure Calls- Part I
    - Sockets
    - Remote Invocations
- Announcement:

Communicating Entities in Distributed Systems

- Communicating entities in distributed systems can be classified into two types:
  - System-oriented entities
    - Processes
    - Threads
    - Nodes
  - Problem-oriented entities
    - Objects (in *object-oriented programming* based approaches)

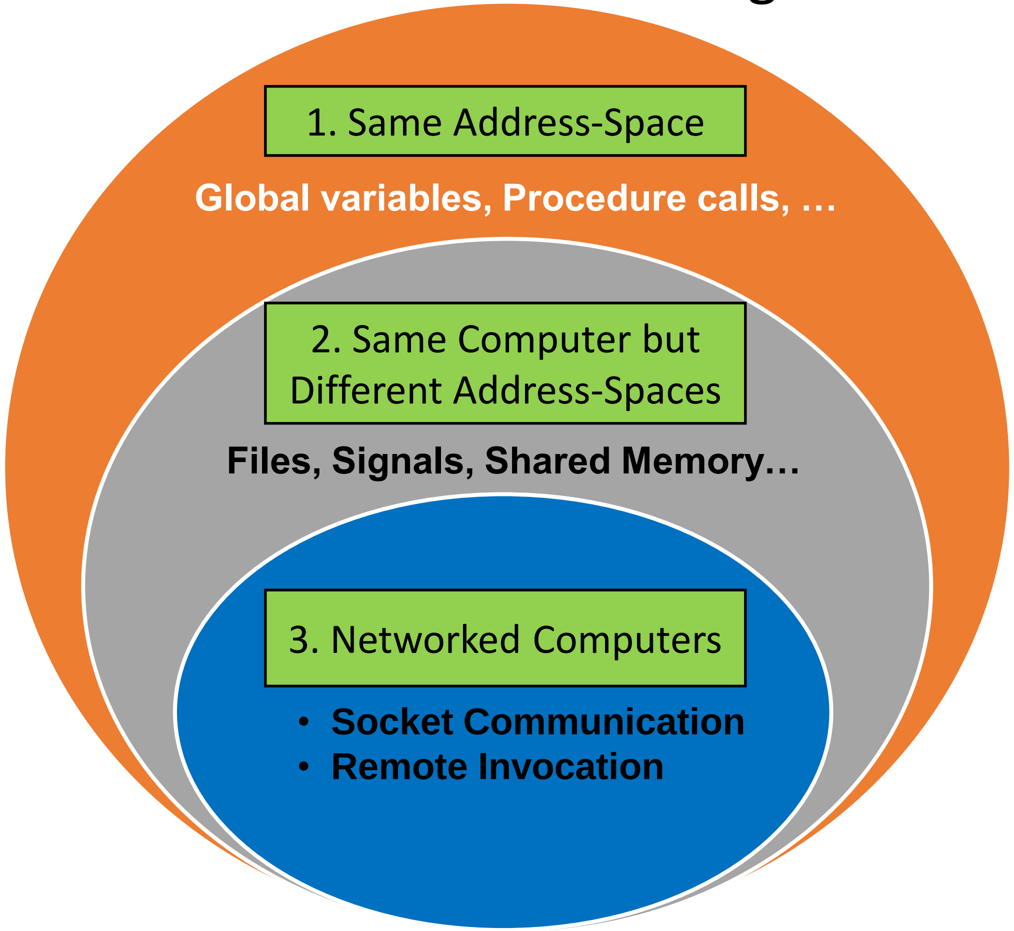
How can entities in distributed systems communicate?

Communication Paradigms

- Communication paradigms describe and classify a set of methods by which entities can interact and exchange data

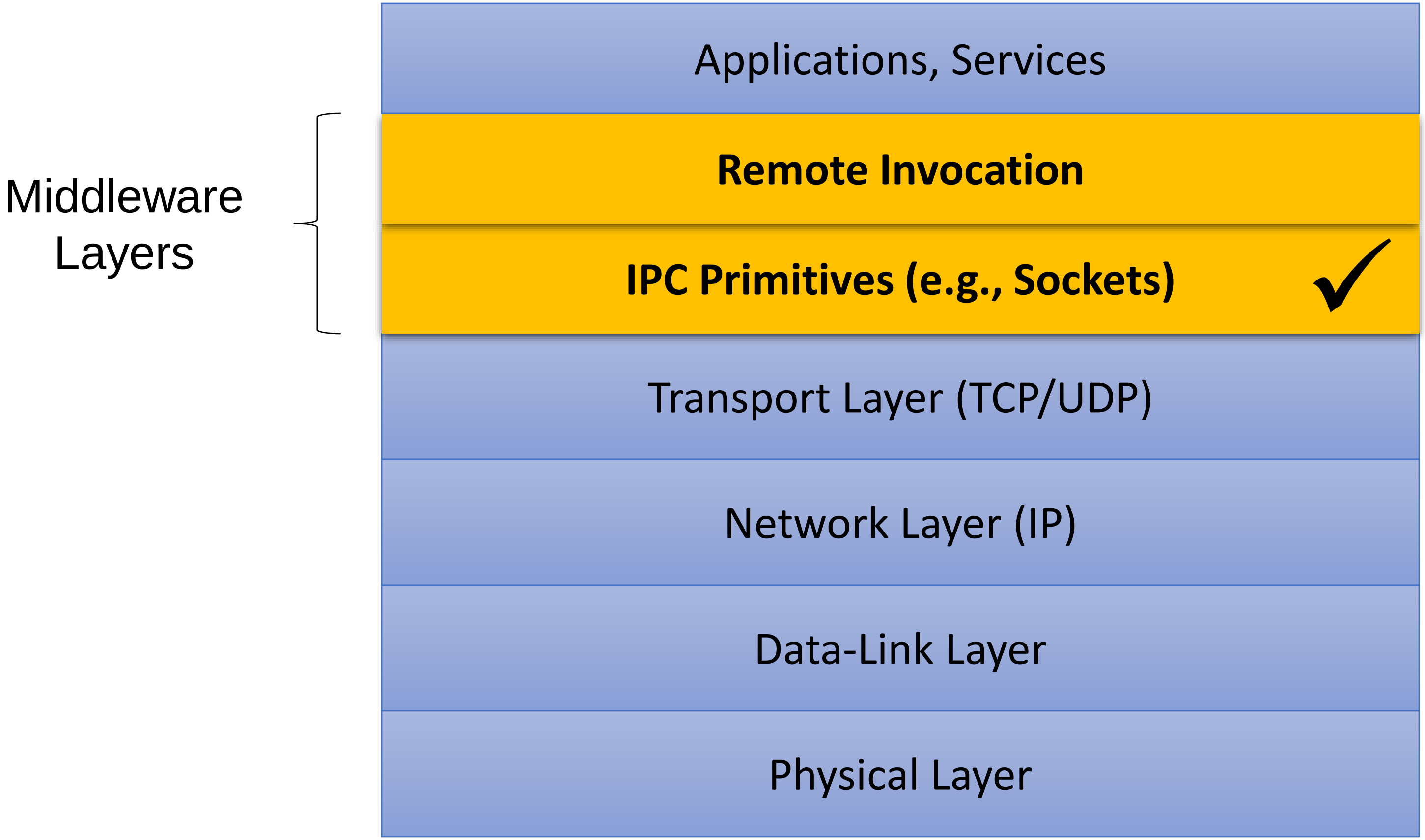
Classification of Communication Paradigms

- Communication paradigms can be categorized into *three* types based on where the entities reside. If entities are running on:



Today, we will study how entities that reside on networked computers communicate in distributed systems using socket communication and remote invocation

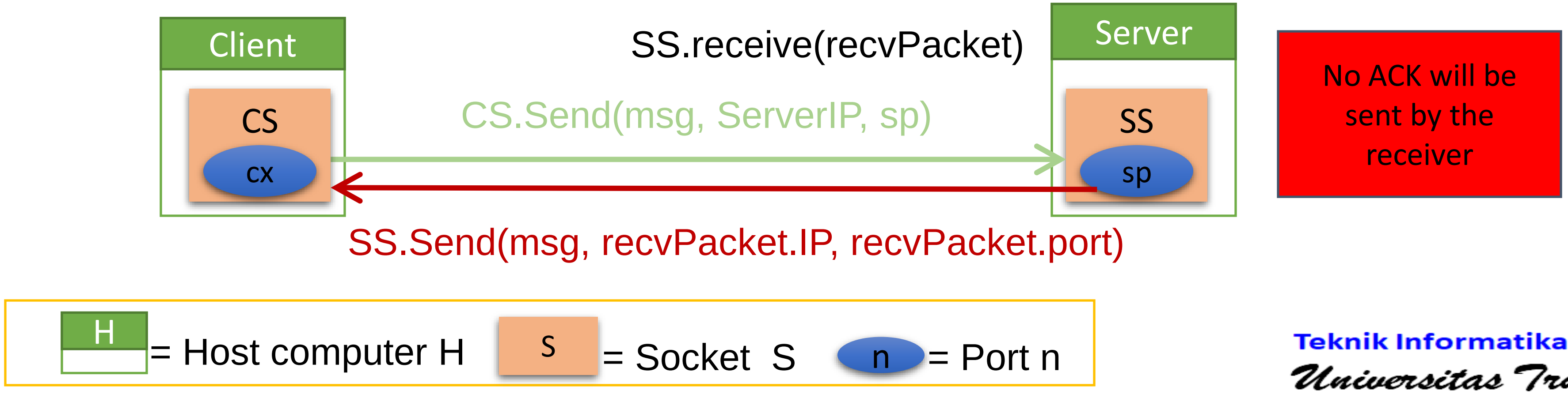
Middleware Layers



UDP Sockets

- UDP provides *connectionless* communication, with no acknowledgements or message retransmissions

- Communication mechanism:
  - Server opens a UDP socket *SS* at a known port *sp*,
  - Socket *SS* waits to receive a request
  - Client opens a UDP socket *CS* at a random port *cx*
  - Client socket *CS* sends a message to ServerIP and port *sp*
  - Server socket SS may send back data to *CS*

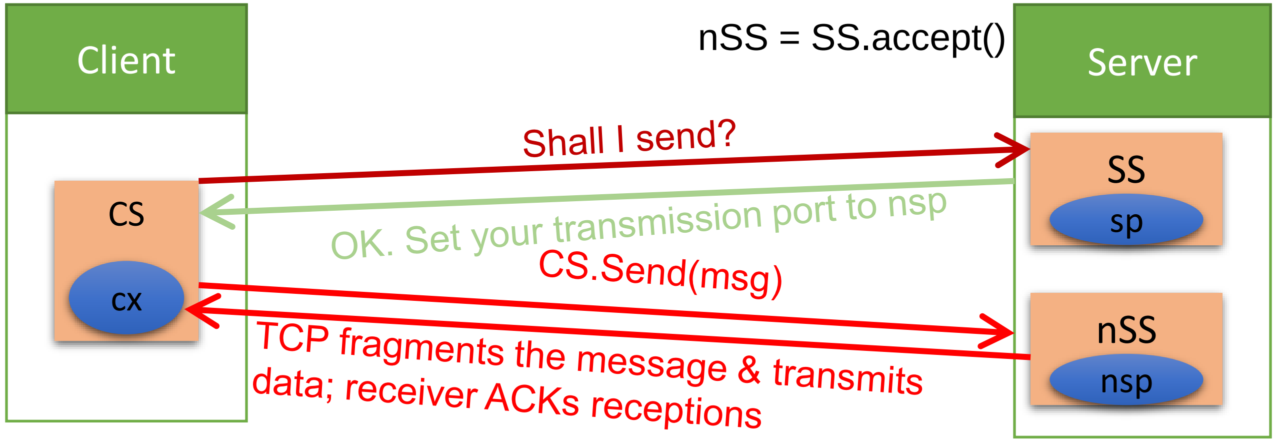


UDP– Design Considerations

- Sender must explicitly fragment a long message into smaller chunks before transmission
  - A maximum size of 548 bytes is suggested for transmission
- Messages may be delivered out-of-order
  - If necessary, programmer must re-order packets
- Communication is not reliable
  - Messages might be dropped due to check-sum errors or buffer overflows at routers
- Receiver should allocate a buffer that is big enough to fit the sender's message
  - Otherwise the message will be truncated

TCP Sockets

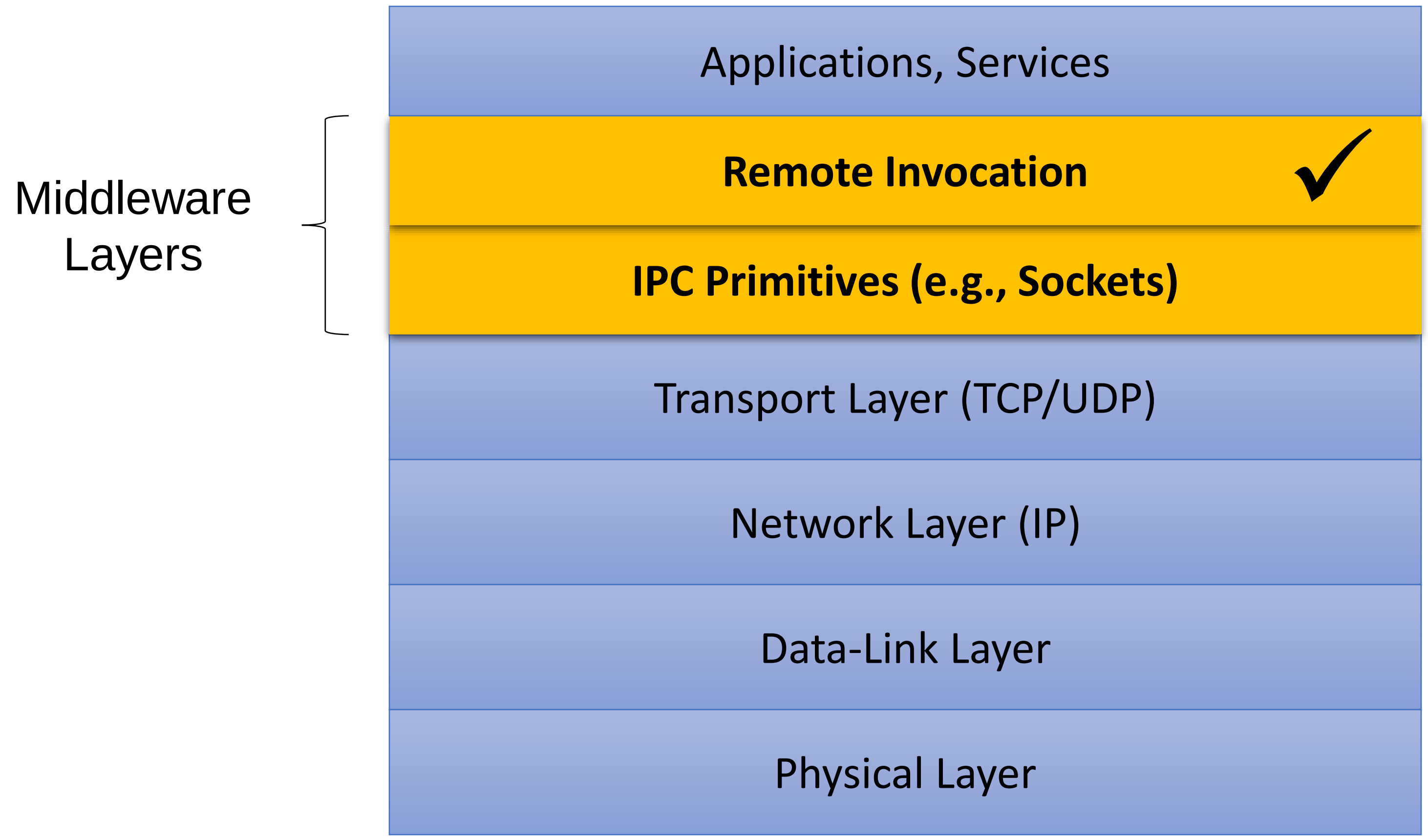
- TCP provides *in-order* delivery, *reliability,* and *congestion control*
- Communication mechanism:
  - Server opens a TCP server socket *SS* at a known port *sp*
  - Server waits to receive a request (using *accept* call)
  - Client opens a TCP socket *CS* at a random port *cx*
  - *CS* initiates a connection initiation message to ServerIP and port *sp*
  - Server socket SS allocates a new socket NSS on random port *nsp* for the client
  - *CS* can send data to *NSS*



Main Advantages of TCP

- TCP ensures in-order delivery of messages
- Applications can send messages of any size
- TCP ensures *reliable communication* via using acknowledgements and retransmissions
- Congestion control of TCP regulates sender rate, and thus prevents network overload

Middleware Layers



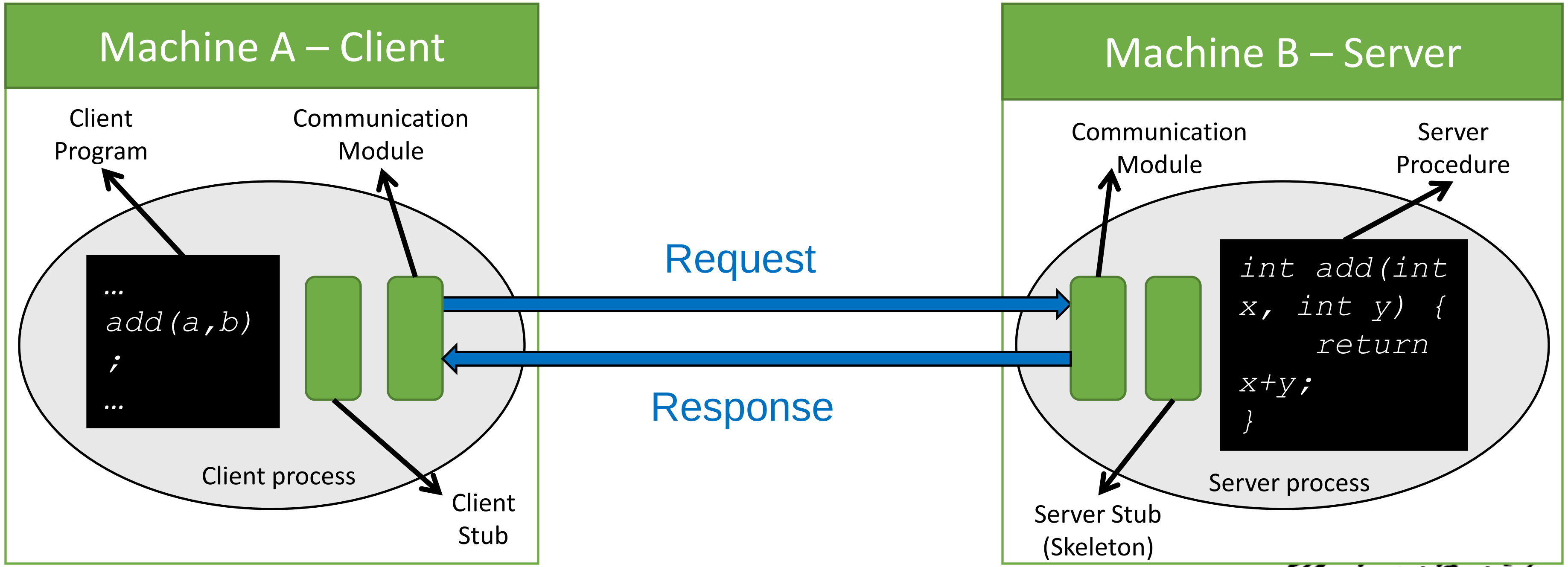
Remote Invocation

- Remote invocation enables an entity to call a procedure that typically executes on an another computer without the programmer explicitly coding the details of communication
  - The underlying middleware will take care of raw-communication
  - Programmer can *transparently* communicate with remote entity

- We will study two types of remote invocations:
  a. Remote Procedure Calls (RPC)
  b. Remote Method Invocation (RMI)

Remote Procedure Calls (RPC)

- RPC enables a sender to communicate with a receiver using a simple procedure call
  - No communication or message-passing is visible to the programmer
- Basic RPC Approach:



Client Stub

- The client stub:
  - Gets invoked by user code as a local procedure
  - *Packs* (or *serializes* or *marshals*) parameters into a request packet (say, request-pkt)
  - Invokes a client side transport routine (e.g., makerpc(request-pkt, &reply-pkt))
  - *Unpacks* (or *de-serializes* or *unmarshals*) reply-pkt into output parameters
  - Returns to user code

Server Stub

- The server stub:
  - Gets invoked after a server side transport routine (e.g., getrequest()) is returned
  - Unmarshals arguments, de-multiplexes opcode, and invokes local server code
  - Marshals arguments, invokes a server-side transport routine (e.g., sendresponse()), and returns to server loop
    - E.g., Typical server main loop:

```
while (1) {
        get-request (&p);      /* blocking call */
        execute-request (p);  /* demux based on opcode */
}
```

Challenges in RPC

- Parameter passing via marshaling
  - Procedure parameters and results have to be transferred over the network as bits
- Data representation
  - Data representation has to be uniform
    - Architecture of the sender and receiver machines may differ
- Failure Independence
  - Client and server might fail independently

Challenges in RPC

- Parameter passing via marshaling
  - Procedure parameters and results have to be transferred over the network as bits
- Data representation
  - Data representation has to be uniform
    - Architecture of the sender and receiver machines may differ
- Failure Independence
  - Client and server might fail independently

Parameter Passing via Marshaling

- Packing parameters into a message that will be transmitted over the network is called *parameter marshalling*
- The parameters to the procedure and the result have to be marshaled before transmitting them over the network
- Two types of parameters can be passed:
    1. Value parameters
    2. Reference parameters

1. Passing Value Parameters

- Value parameters have complete information about the variable, and can be directly encoded into the message
  - E.g., integer, float, character
- Values are passed through call-by-value
  - The changes made by the callee procedure are not reflected in the caller procedure

2. Passing Reference Parameters

- Passing reference parameters like value parameters in RPC leads to incorrect results due to two reasons:

  a. Invalidity of reference parameters at the server
     - Reference parameters are valid only within client's address space
     - Solution: Pass the reference parameter by copying the data that is referenced

  b. Changes to reference parameters are not reflected back at the client
     - Solution: "Copy/Restore" the data
       - Copy the data that is referenced by the parameter
       - Copy-back the value at server to the client

Challenges in RPC

- Parameter passing via marshaling
  - Procedure parameters and results have to be transferred over the network as bits
- Data representation
  - Data representation has to be uniform
    - Architecture of the sender and receiver machines may differ
- Failure Independence
  - Client and server might fail independently

Data Representation

- Computers in DSs often have different architectures and operating systems
  - The size of the data-type differ
    - E.g., A *long* data-type is 4-bytes in 32-bit Unix, while it is 8-bytes in 64-bit Unix systems
  - The format in which the data is stored differs
    - E.g., Intel stores data in little-endian format, while SPARC stores in big-endian format
- The client and server have to agree on how simple data is represented in the message
  - E.g., Format and size of data-types such as integer, char and float

Challenges in RPC

- Parameter passing via marshaling
  - Procedure parameters and results have to be transferred over the network as bits
- Data representation
  - Data representation has to be uniform
    - Architecture of the sender and receiver machines may differ
- Failure Independence
  - Client and server might fail independently

Failure Independence

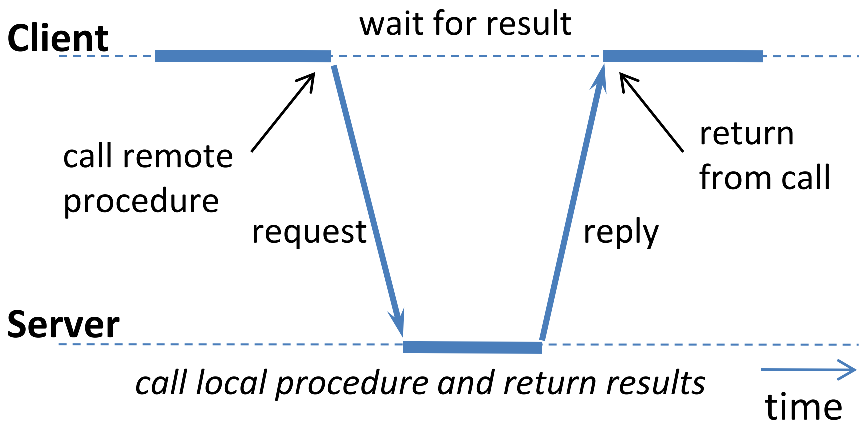
- In the local case, the client and server live or die together
- In the remote case, the client sees new *failure types* (*more on this next lecture*)
  - Network failure
  - Server machine crash
  - Server process crash
- Thus, failure handling code has to be more thorough (and essentially more complex)

Remote Procedure Call Types

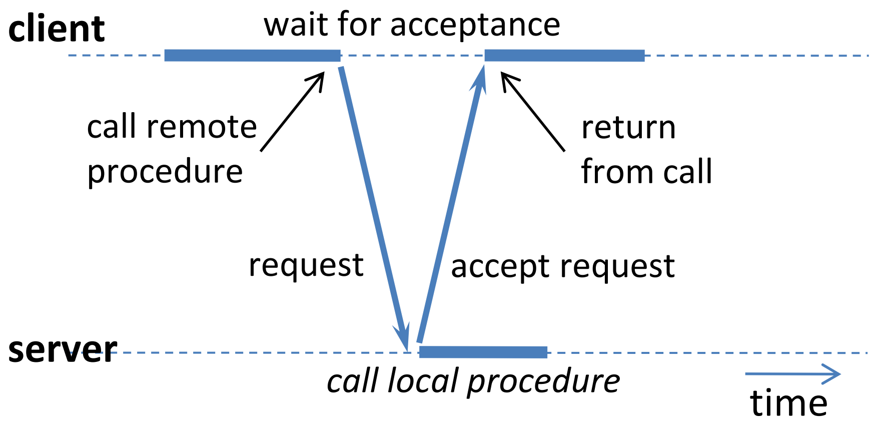
- Remote procedure calls can be:
  - Synchronous
  - Asynchronous (or Deferred Synchronous)

Synchronous vs. Asynchronous RPCs

- An RPC with strict request-reply blocks the client until the server returns
  - Blocking wastes resources at the client
- Asynchronous RPCs are used if the client does not need the result from server
  - The server immediately sends an ACK back to the client
  - The client continues the execution after an ACK from the server



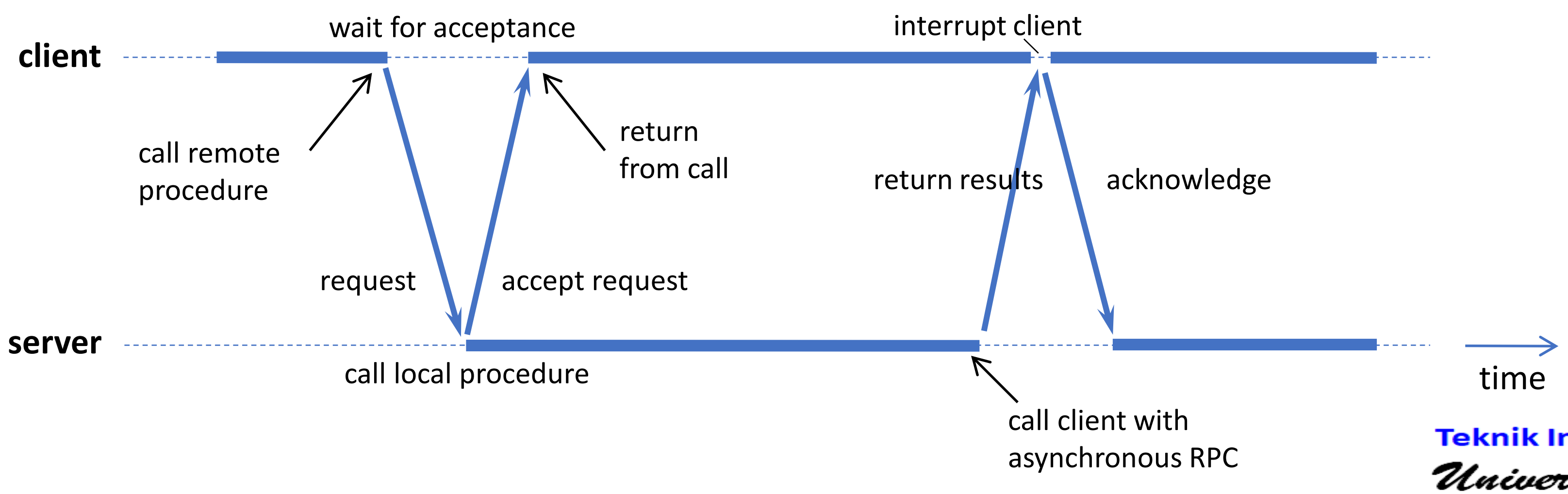
Synchronous RPCs



Asynchronous RPCs

Deferred Synchronous RPCs

- Asynchronous RPC is also useful when a client wants the results, but does not want to be blocked until the call finishes
- Client uses *deferred synchronous* RPCs
  - Single request-response RPC is split into two RPCs
  - First, client triggers an asynchronous RPC on server
  - Second, on completion, server calls-back client to deliver the results



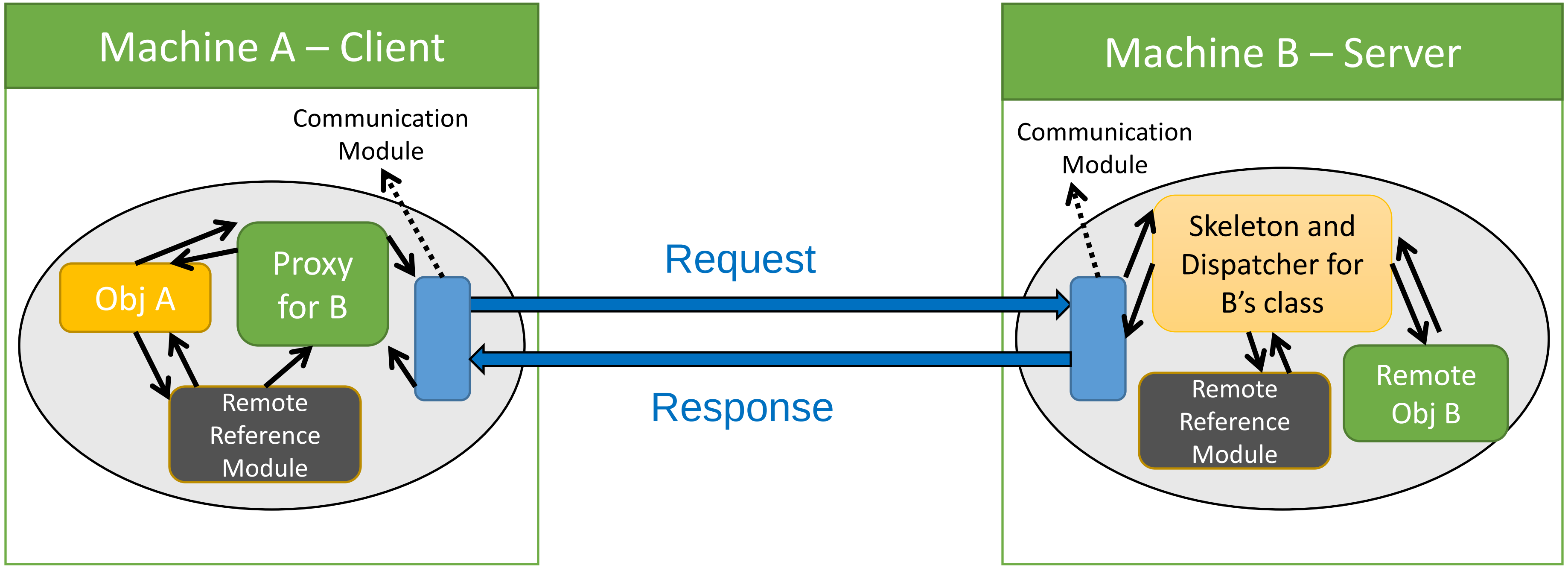
Remote Method Invocation (RMI)

- RMI is similar to RPC, but in a world of distributed objects
  - The programmer can use the full expressive power of object-oriented programming
  - RMI not only allows to pass value parameters, but also pass object references
- In RMI, a calling object can invoke a method on a potentially remote object

Remote Objects and Supporting Modules

- In RMI, objects whose methods can be invoked remotely are known as “*remote objects*”
  - Remote objects implement remote interfaces
- During any method call, the system has to resolve whether the method is being called on a local or a remote object
  - Local calls should be called on a local object
  - Remote calls should be called via remote method invocation
  - *Remote Reference Module* is responsible for translating between local and remote object references

RMI Control Flow



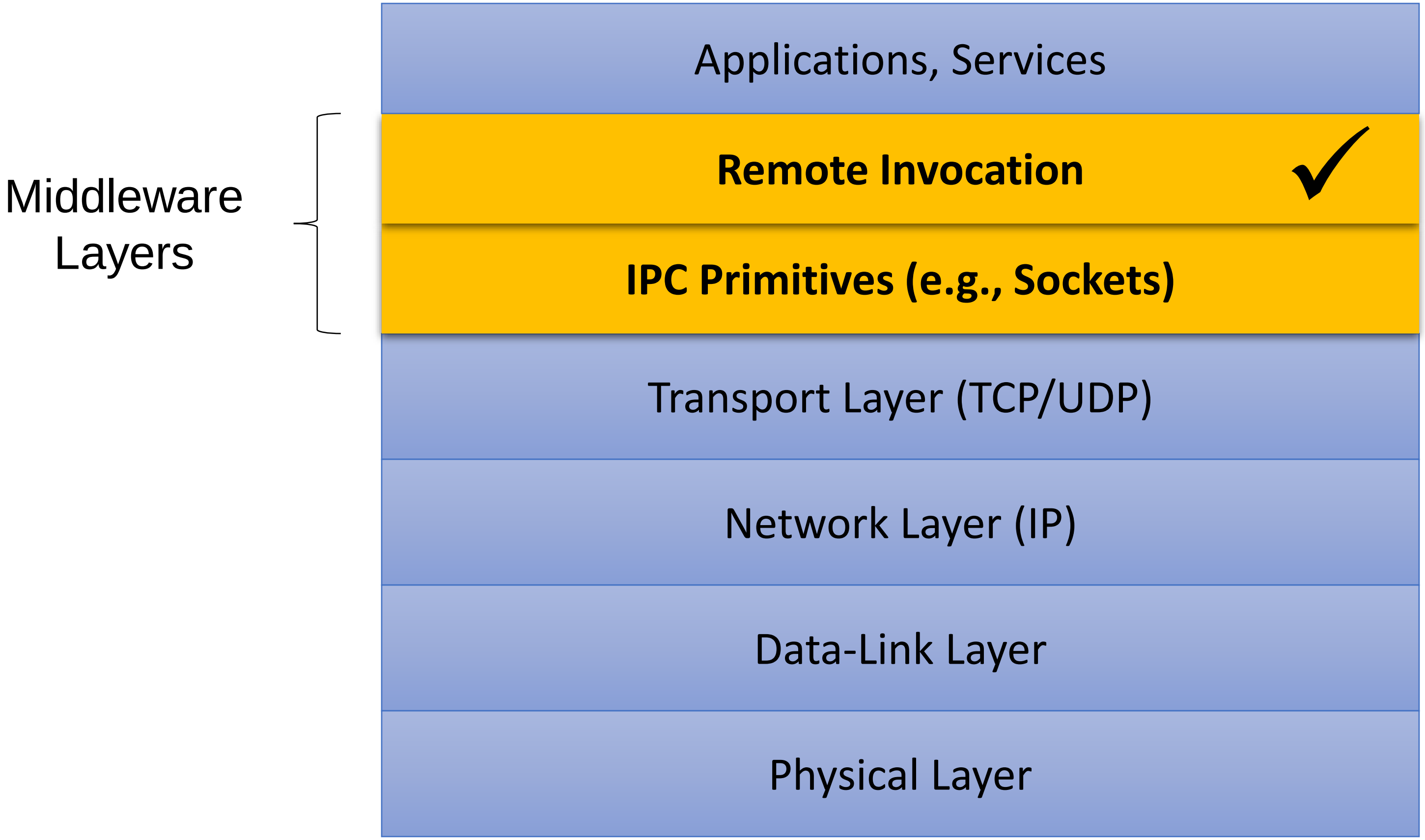
Today…

- Last Session:
  - RPC- Part I

- Today's Session:
  - Continue with Remote Procedure Calls

- Announcement:
  - Project I is now out. It is due on Feb 21 (design report is due on Feb 6)

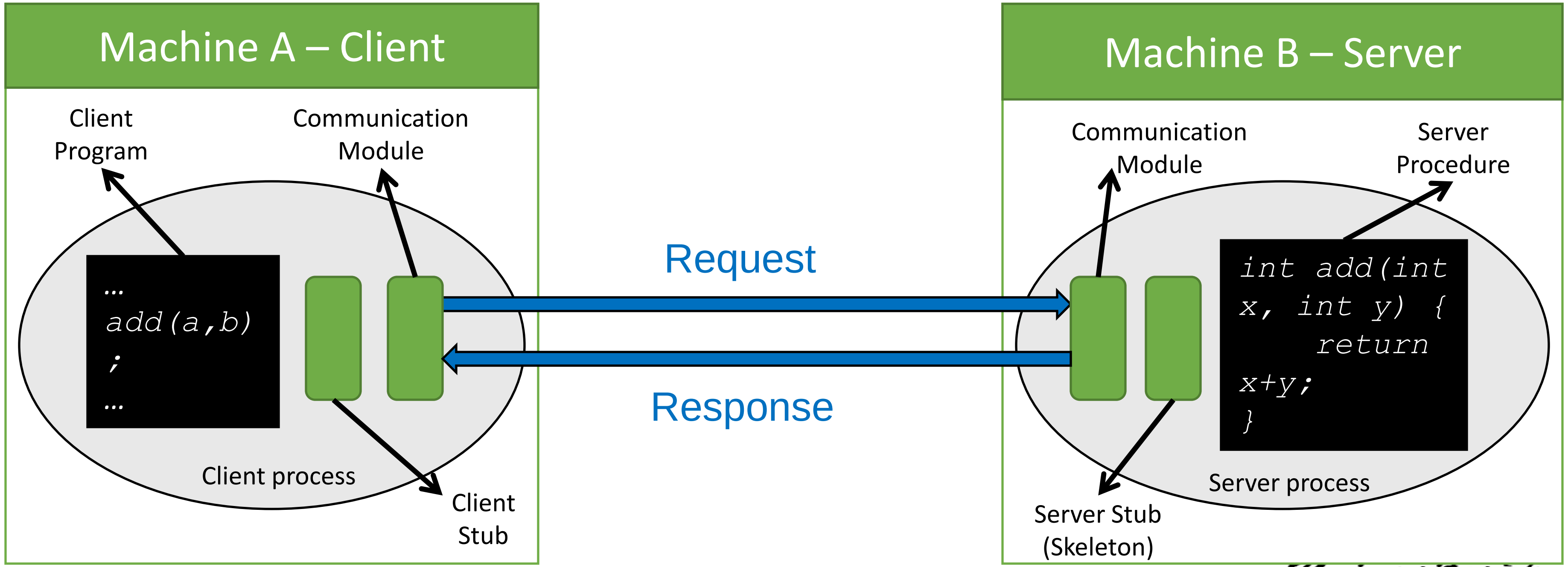
Middleware Layers



Remote Procedure Calls (RPC)

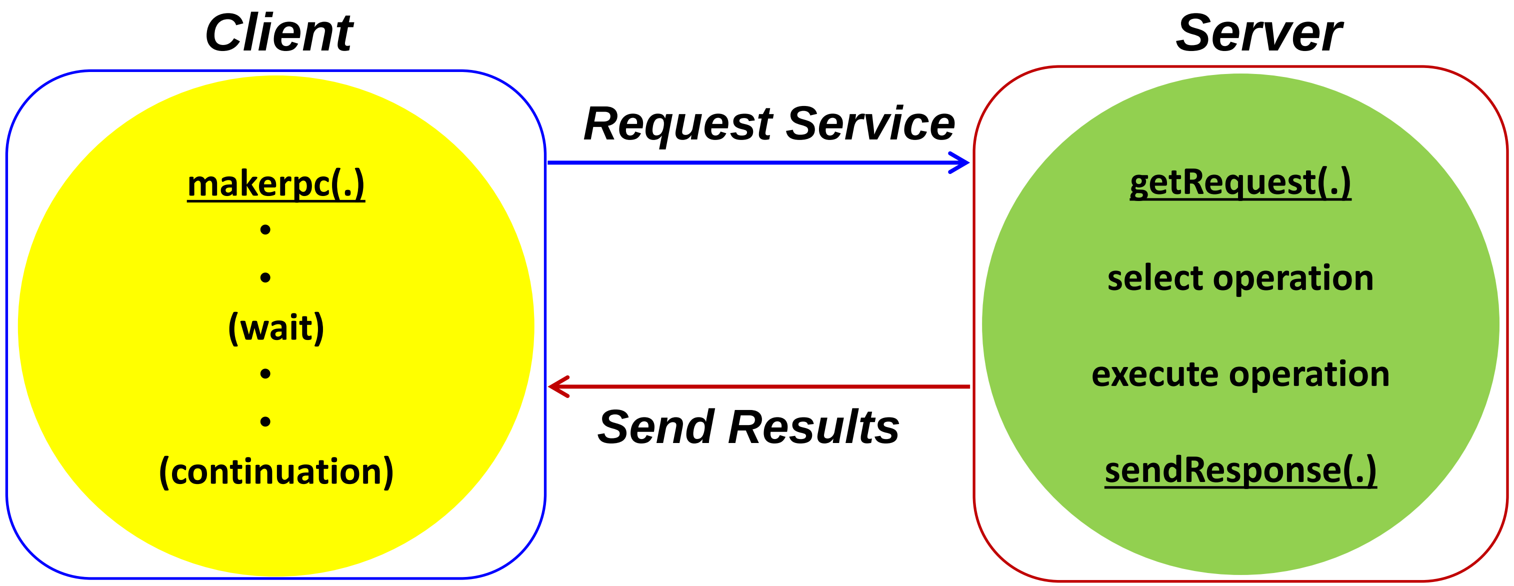
- RPC enables a sender to communicate with a receiver using a simple procedure call
  - No communication or message-passing is visible to the programmer

- Basic RPC Approach:



Transport Primitives

- RPC communication module (or *transport*) is mainly based on a trio of communication primitives, *makerpc(.)*, *getRequest(.)*, and *sendResponse(.)*



Failure Types

- RPC systems may suffer from various types of failures

Type of Failure	Description
• Crash Failure	• A server halts, but was working correctly until it stopped

• Omission Failure
  • Receive Omission
  • Send Omission
	• A server fails to respond to incoming requests
	  • A server fails to receive incoming messages
	  • A server fails to send messages

• Timing Failure	• A server's response lies outside the specified time interval

• Response Failure
  • Value Failure
  • State Transition Failure
	• A server's response is incorrect
	  • The value of the response is wrong
	  • The server deviates from the correct flow of control

• Byzantine Failure	• A server may produce arbitrary responses at arbitrary times

Timeout Mechanism

- To allow for occasions where a request or a reply message is lost, *makerpc(.)* can use a *timeout mechanism*

- There are various options as to what *makerpc(.)* can do after a timeout:
  - Either return immediately with an indication to the client that the request has failed
  - Or *retransmit* the request repeatedly until either a reply is received or the server is assumed to have failed

- How to pick a timeout value?
  - At best, use empirical/theoretical statistics
  - At worst, no good value exists

Idempotent Operations

- In cases when the request message is retransmitted, the server may receive it *more than once*
- This can cause an operation to be executed more than once for the same request
- *Caveat:* *Not* every operation can be executed more than once and obtain the same result each time!
- Operations that CAN be executed repeatedly with the same effect are called *idempotent operations*

Duplicate Filtering

- To avoid problems with operations, the server should:
  - Identify successive messages from the "same" client
    - Monotonically increasing *sequence numbers* can be used
  - Filter out duplicates

- Upon receiving a "duplicate" request, the server can:
  - Either re-execute the operation again and reply
    - Possible only for idempotent operations

  - Or avoid re-executing the operation via *retaining* its output in a non-volatile history (or *log*) file
    - Might necessitate *transactional semantics* *(more on this later in the course)*

Implementation Choices

- RPC transport can be implemented in different ways to provide different *delivery guarantees*. The main choices are:
    1. Retry request service (*client side*): Controls whether to retransmit the request service until either a reply is received or the server is assumed to have failed
    2. Duplicate filtering (*server side*): Controls when retransmissions are used and whether to filter out duplicate requests at the server
    3. Retention of results (*server side*): Controls whether to keep a history of result messages so as to enable lost replies to be retransmitted without re-executing the operations at the server

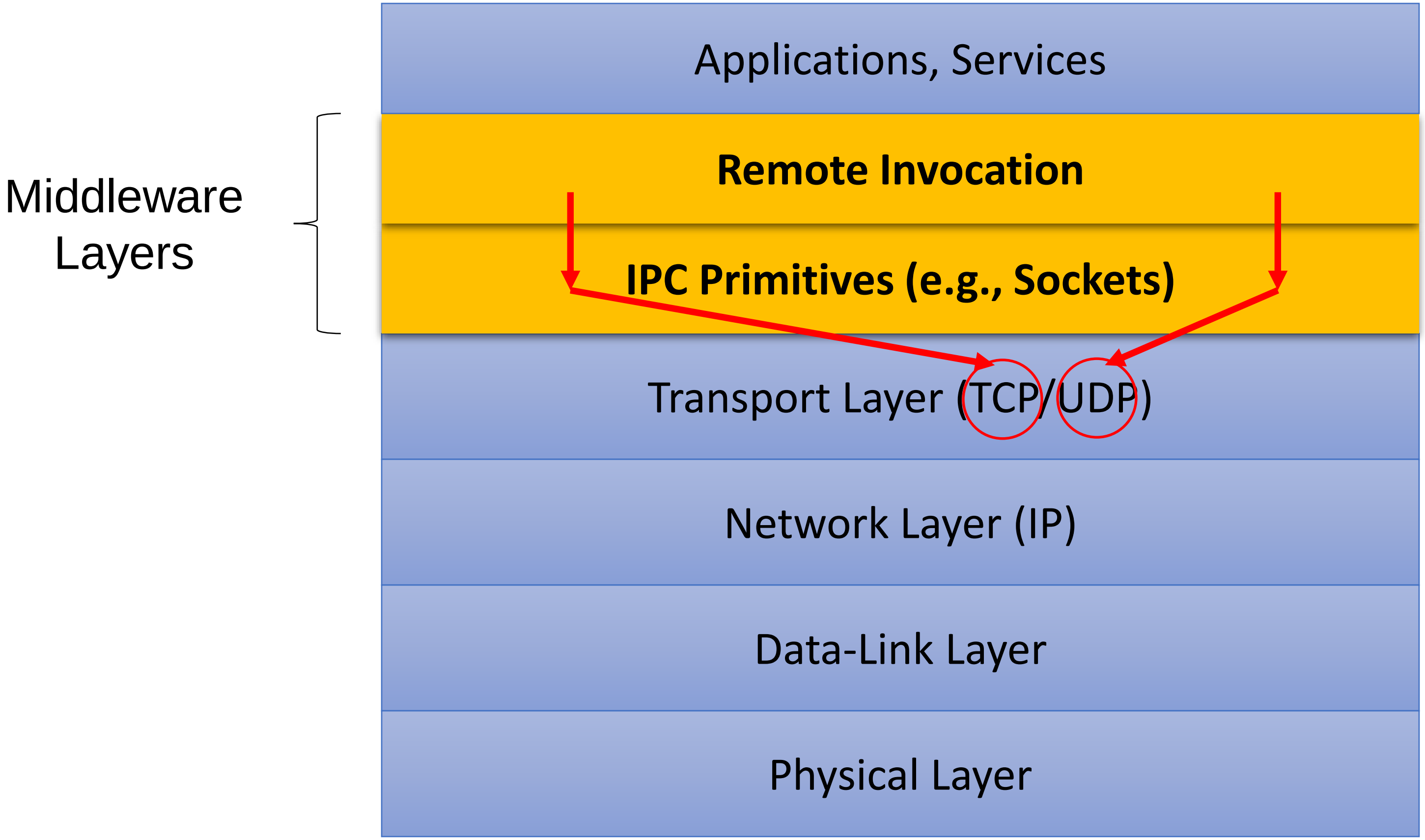
RPC Call Semantics

- Combinations of measures lead to a variety of possible *semantics* for the reliability of RPC

Fault Tolerance Measure			Call Semantics (Pertaining to Remote Procedures)
Retransmit Request Message	**Duplicate Filtering**	**Re-execute Procedure or Retransmit Reply**	
No	N/A	N/A	***Maybe***
Yes	No	Re-execute Procedure	***At-least-once***
Yes	Yes	Retransmit Reply	***At-most-once***

Ideally, we would want an *exactly-once* semantic!

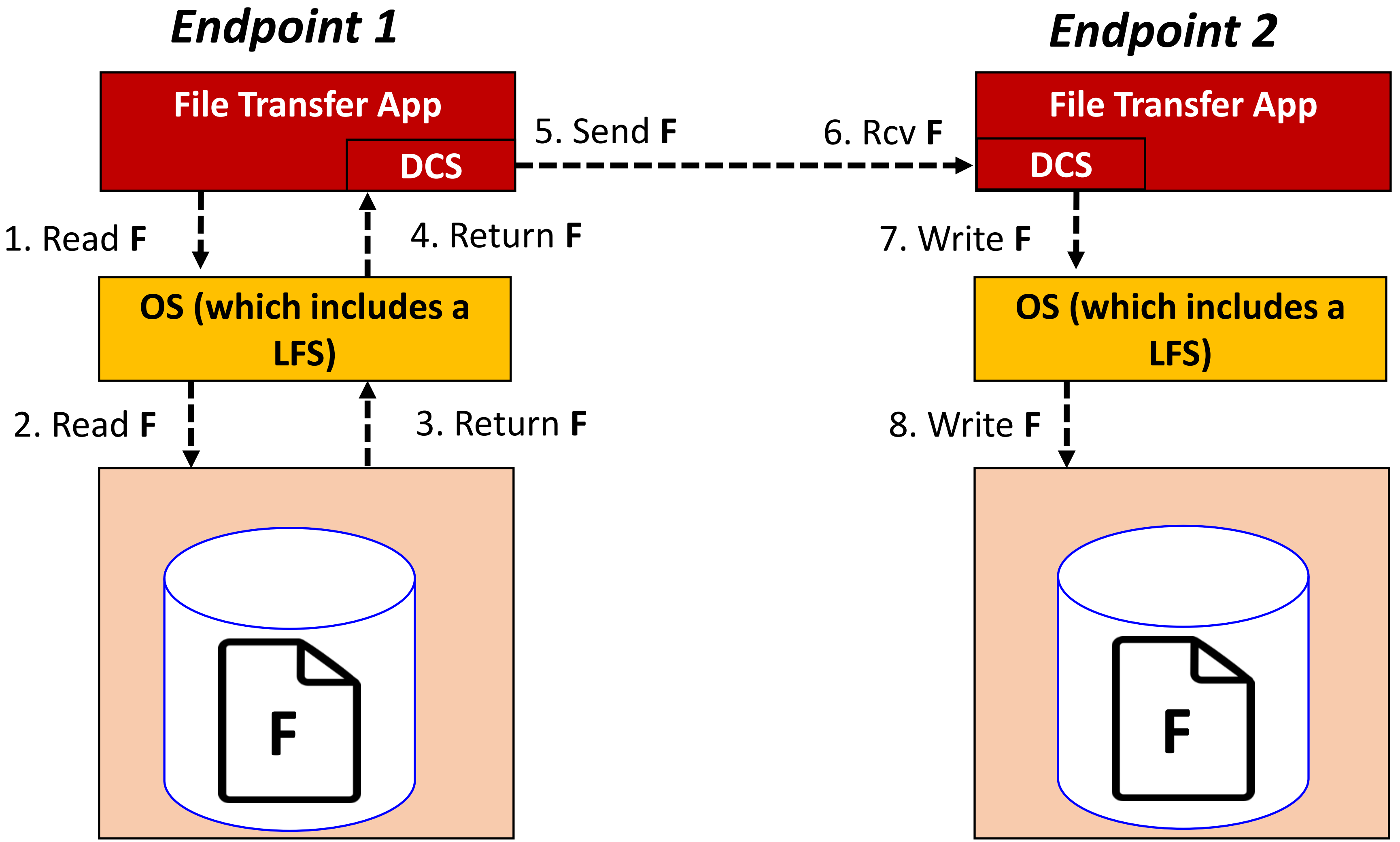
Middleware Layers



RPC over UDP or TCP

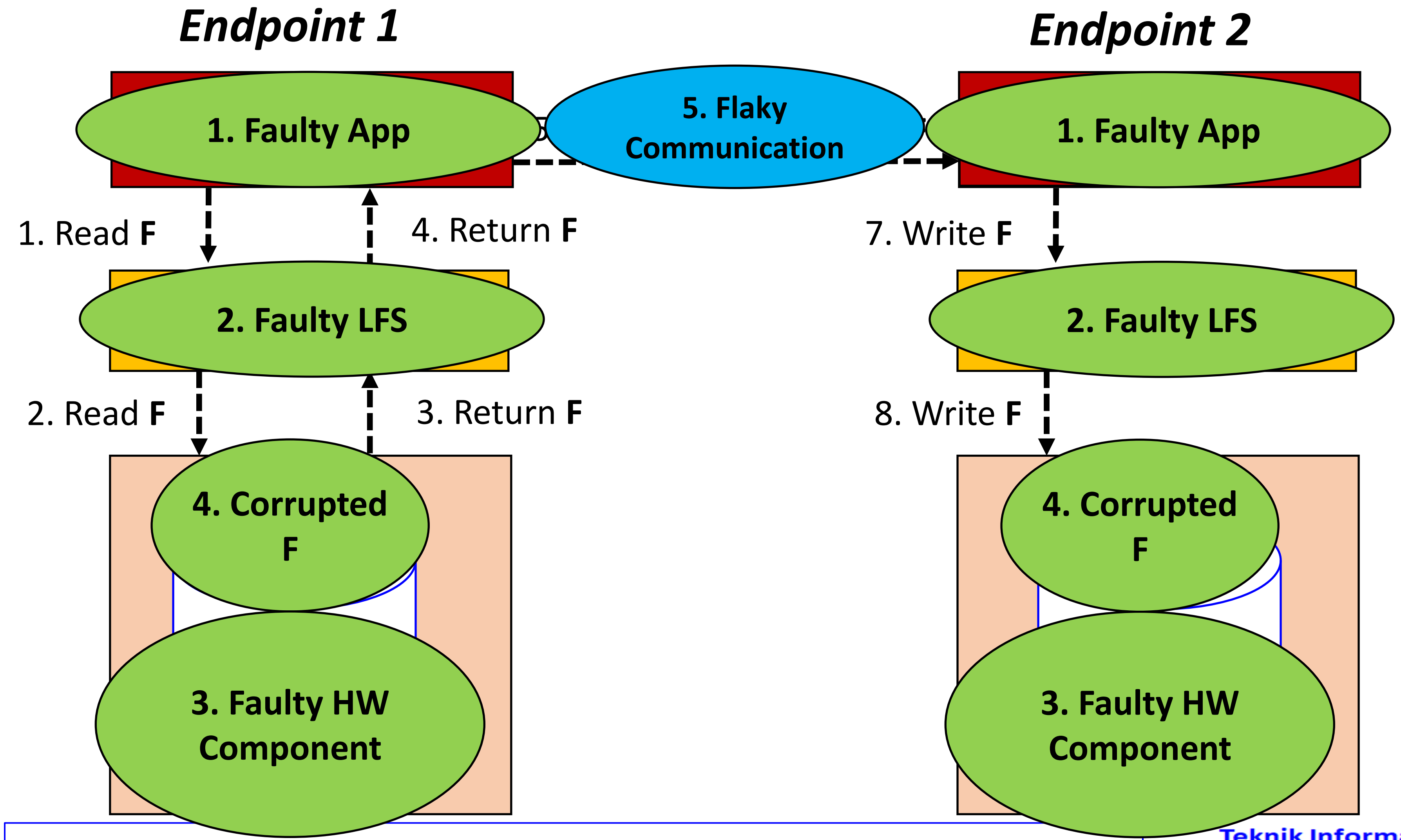
- If RPC is layered on top of UDP
  - Retransmission shall/can be handled by RPC
- If RPC is layered on top of TCP
  - Retransmission will be handled by TCP
  - Is it still necessary to take fault-tolerance measures within RPC?
    - Yes-- read “End-to-End Arguments in System Design” by Saltzer *et. al.*

Careful File Transfer: Flow



DCS = Data Communication System; LFS = Local File System

Careful File Transfer: Possible Threats



DCS = Data Communication System; LFS = Local File System

Careful File Transfer: End-To-End Check and Retry

- Endpoint 1 stores with F a checksum C_A
- After Endpoint 2 writes F, it reads it again from disk, calculates a checksum C_B, and sends it back to Endpoint 1
- Endpoint 1 compares C_A and C_B
  - If $C_A = C_B$, commit the file transfer
  - Else, retry the file transfer

Careful File Transfer: End-To-End Check and Retry

- How many retries?
  - Usually 1 if failures are rare
  - 3 retries might indicate that some part of the system needs repair
- What if the Data Communication System uses TCP?
  - Only threat 5 (e.g., packet loss due to a flaky communication) is eliminated
  - The frequency of retries gets reduced if the fault was caused by the communication system
  - More *control* traffic, but only missing parts of F need to be reshipped
  - The file transfer application still needs to apply *end-to-end reliability measures*!

Careful File Transfer: End-To-End Check and Retry

- What if the Data Communication System uses UDP?
  - Threat 5 (e.g., packet loss due to a flaky communication) is NOT eliminated- *F needs to be reshipped by the application if no measures are taken to address this threat*
  - The frequency of retries might increase
  - Worse performance on flaky links
  - *The file transfer application still needs to apply end-to-end reliability measures!*

In *both cases*, the application needs to provide end-to-end reliability guarantees!

Kuliah Berikutnya

- Layanan Penamaan

Pertanyaan?