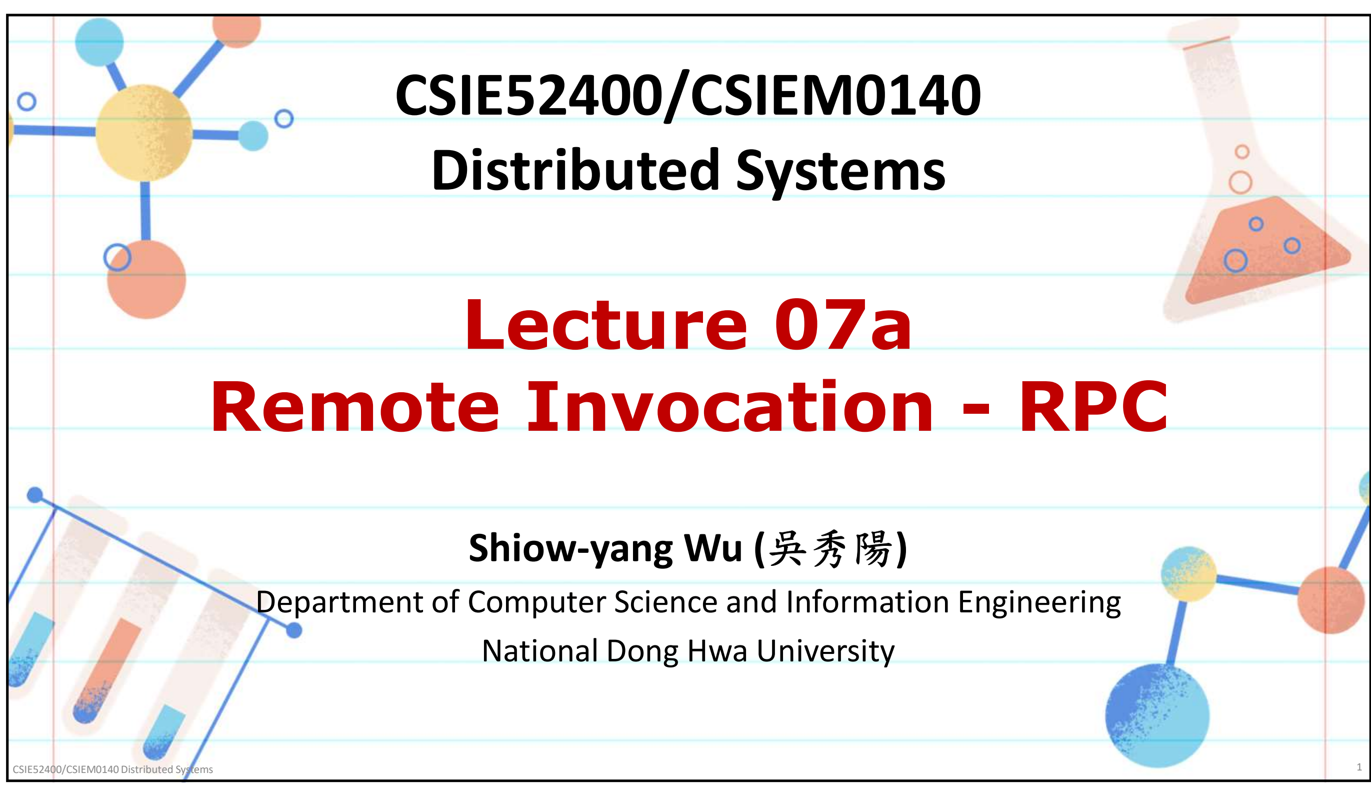
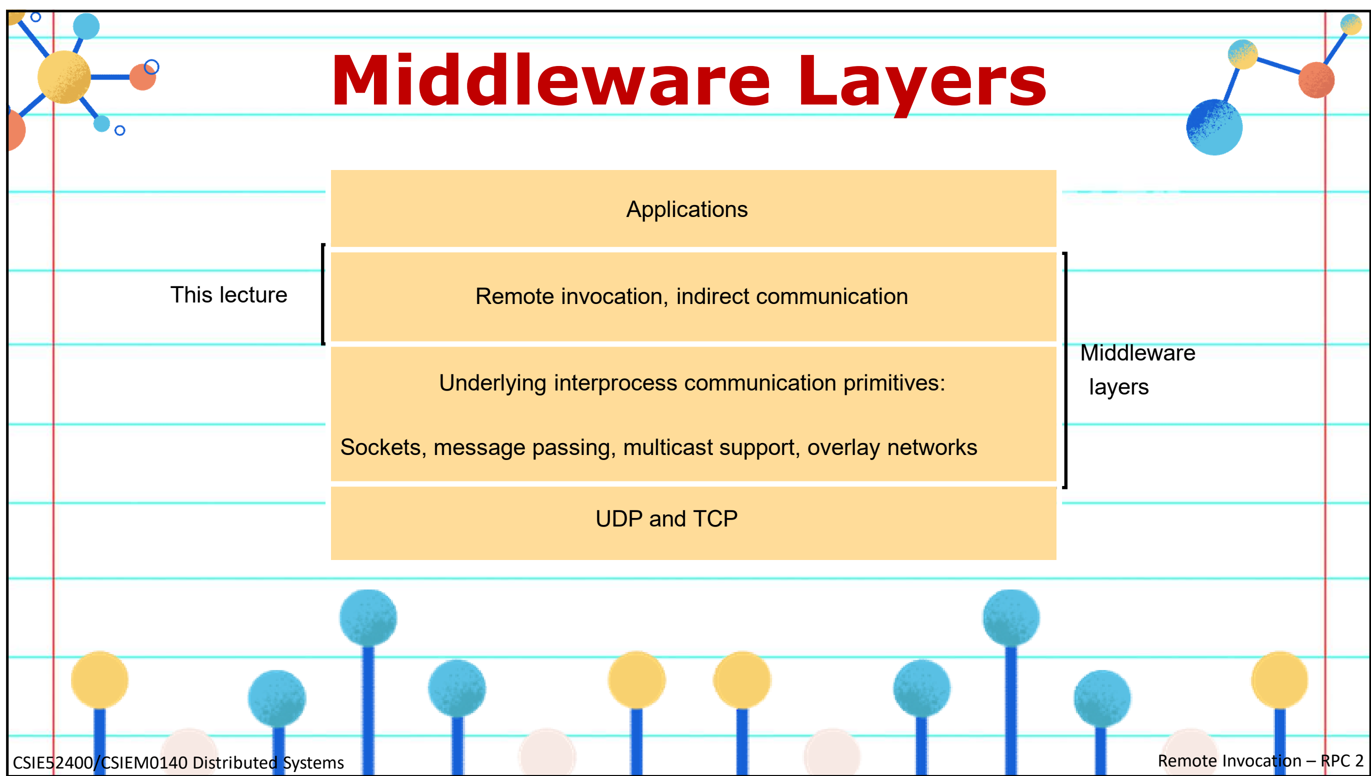
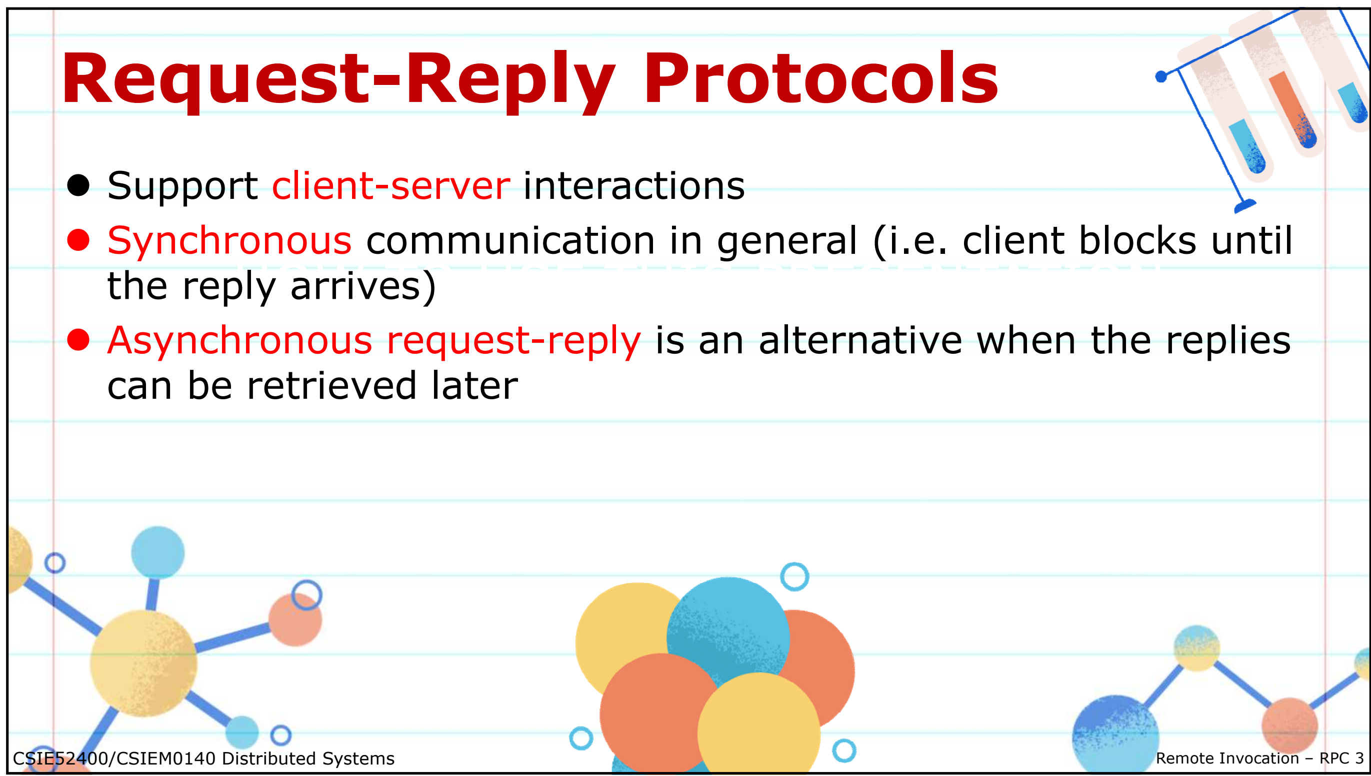
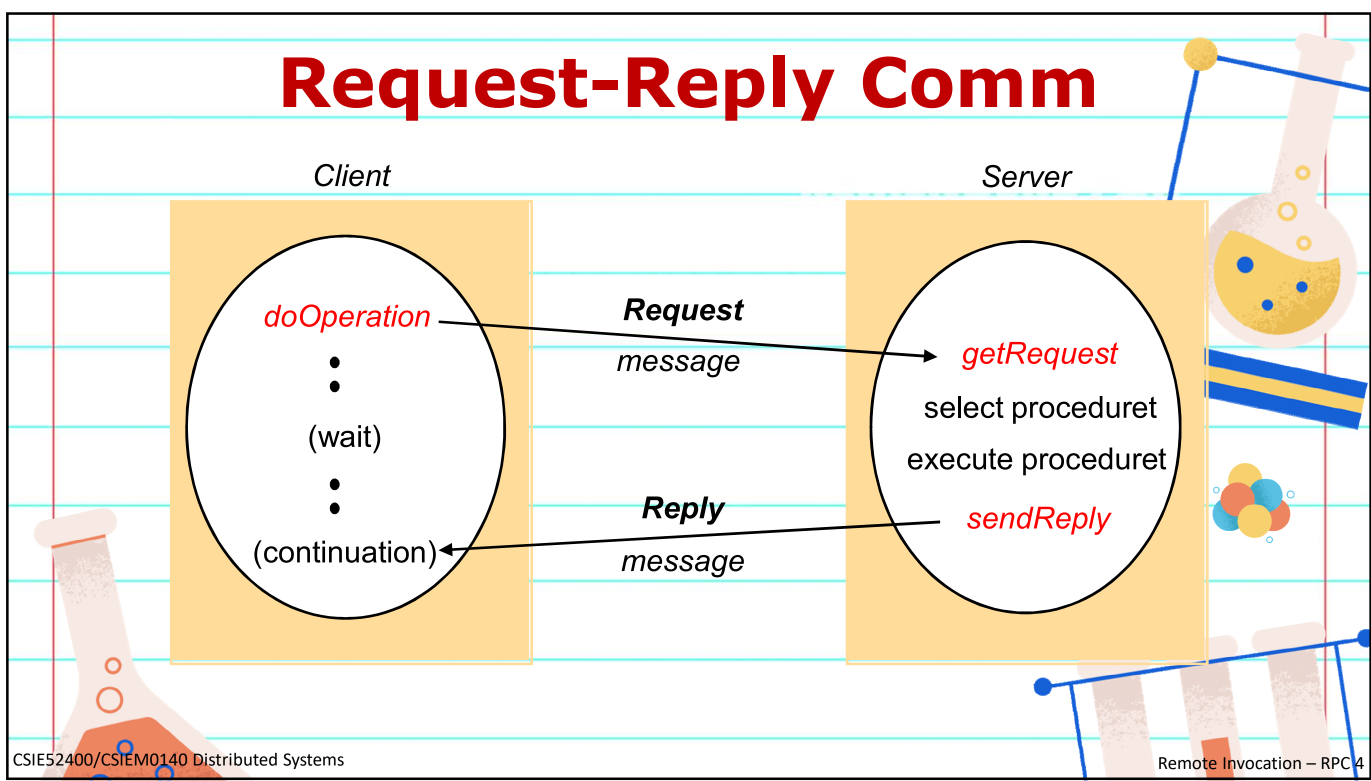
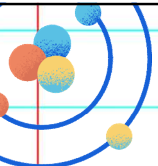

CSIE52400/CSIEM0140
Distributed Systems
Lecture 07a
Remote Invocation - RPC
Shiow-yang Wu (吳秀陽)
Department of Computer Science and Information Engineering
National Dong Hwa University
CSIE52400/CSIEM0140 Distributed Systems
1


Middleware Layers
Applications
This lecture
Remote invocation, indirect communication
Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks
Middleware layers
UDP and TCP
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 2


Request-Reply Protocols
Support client-server interactions
Synchronous communication in general (i.e. client blocks until the reply arrives)
Asynchronous request-reply is an alternative when the replies can be retrieved later
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 3


Request-Reply Comm
Client
Server
doOperation
(wait)
(continuation)
Request
message
Reply
message
getRequest
select proceduret
execute proceduret
sendReply
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 4



Request-Reply Protocol Operations

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.

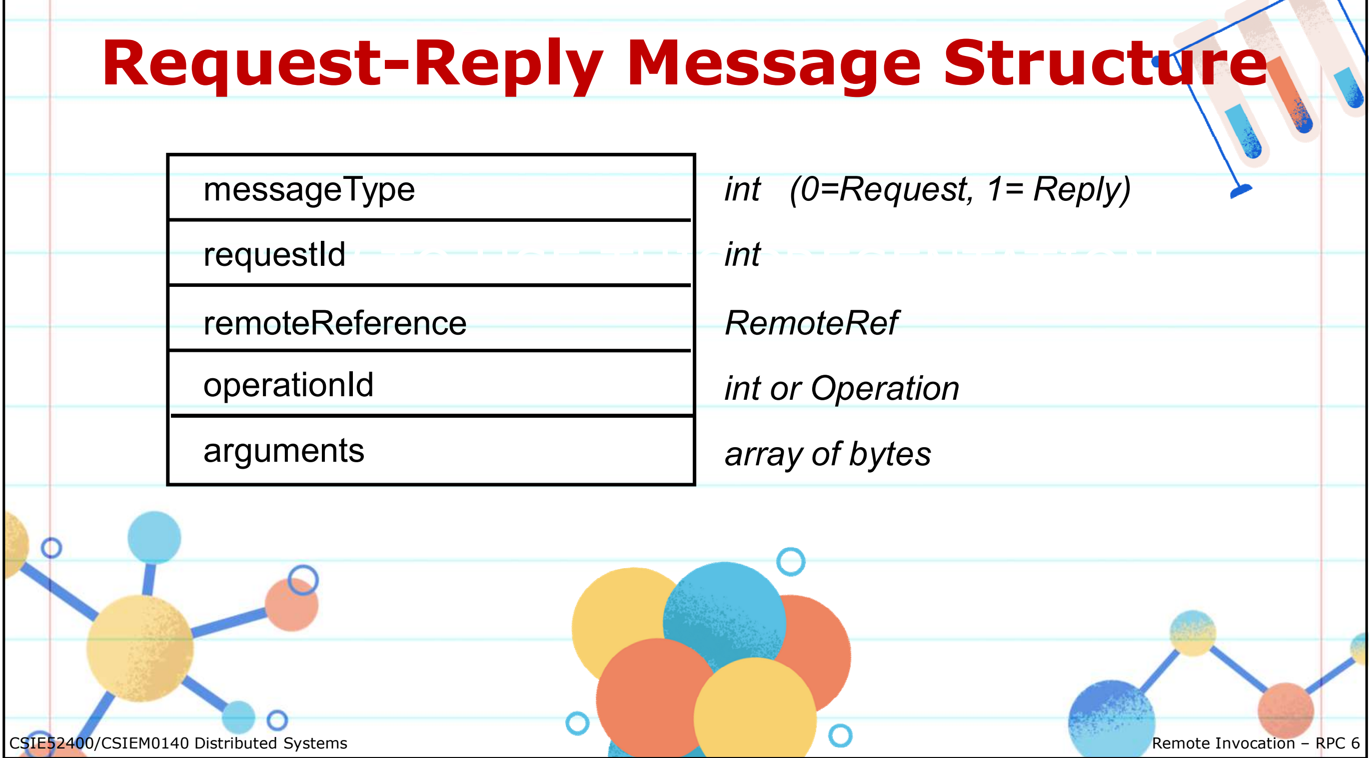
public byte[ ] getRequest ();

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



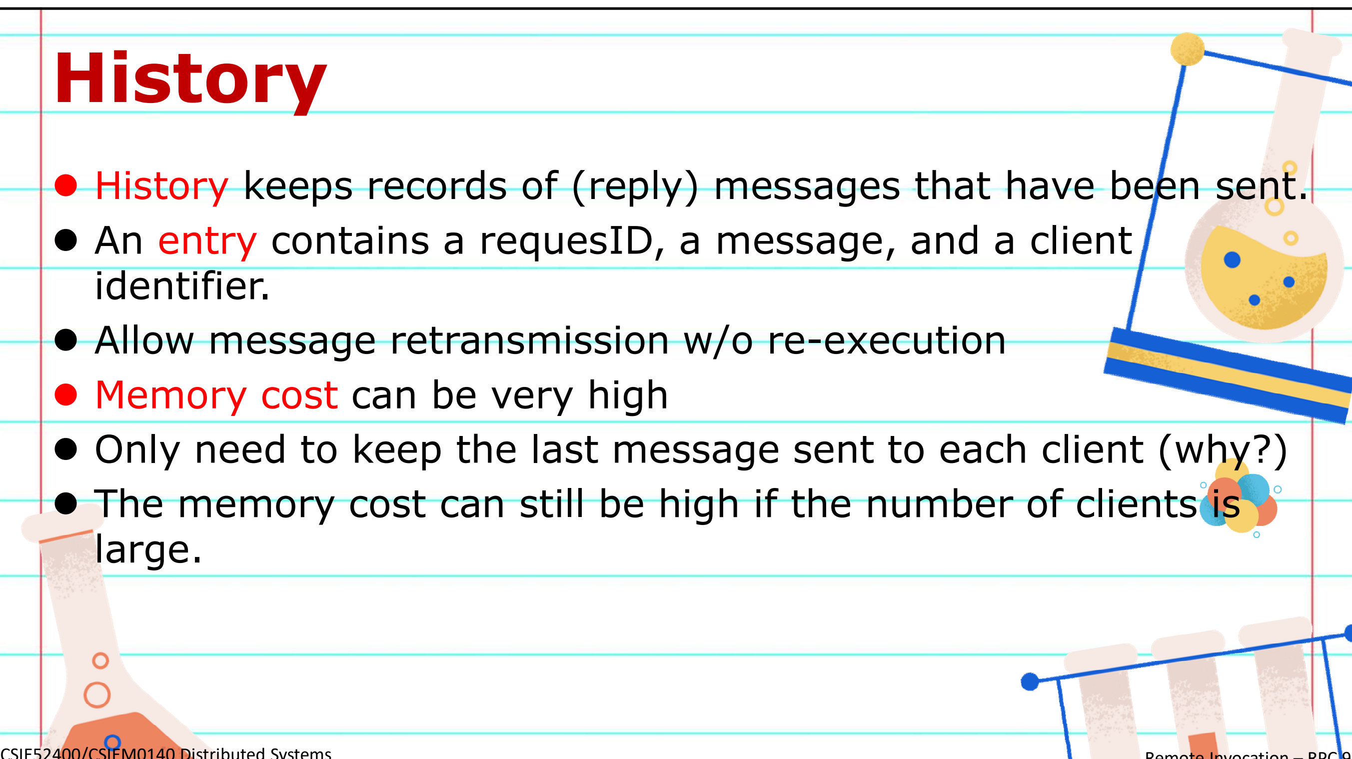
Field	Type
messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*array of bytes*

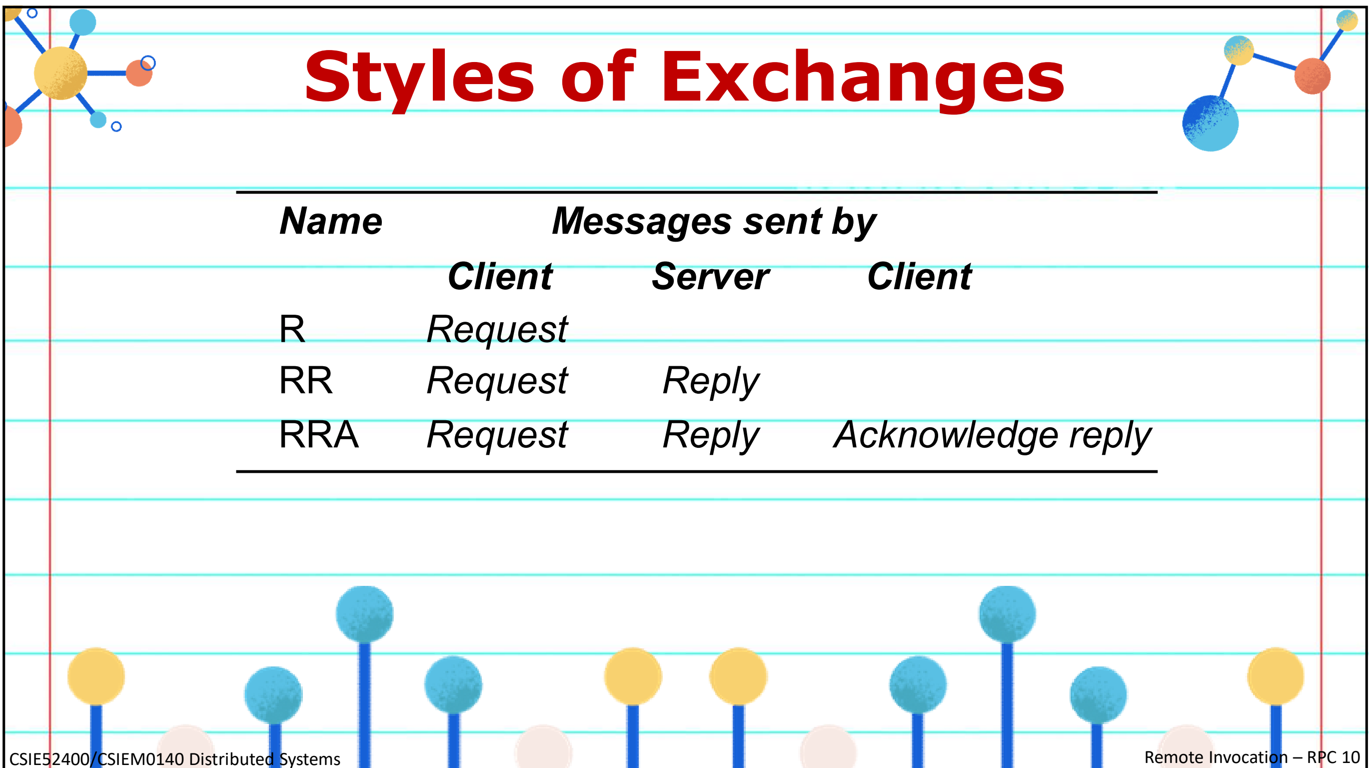
Protocol Details

- A message identifier consists of two parts
  - requestID
  - An identifier of the sender
- Failure model
  - Omission failures
  - Messages may not be delivered in sender order
- Timouts
  - Return immediately with failure (not normal)
  - Resend the request
  - Returns with exception after retries

Protocol Details

- Duplicate requests
  - Server may receive the same request more than once (why?)
  - Recognize successive messages with the same requestID and filter out duplicates
- Lost reply
  - Lost reply may lead to re-execution if the original result was not stored
  - Idempotent operation can be executed repeatedly with the same effect


History
History keeps records of (reply) messages that have been sent.
An entry contains a requesID, a message, and a client identifier.
Allow message retransmission w/o re-execution
Memory cost can be very high
Only need to keep the last message sent to each client (why?)
The memory cost can still be high if the number of clients is large.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 9


Styles of Exchanges
Name
Messages sent by
Client
Server
Client
R
Request
RR
Request
Reply
RRA
Request
Reply
Acknowledge reply
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 10

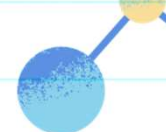
HTTP: An Example

- HyperText Transfer Protocol (HTTP) is an example of a request-reply protocol
- Clients (eg. Browsers) send HTTP requests to servers (eg. Web servers) and wait for HTTP responses.
- Support a fixed set of methods (GET, PUT, POST, etc.) applicable to all server's resources.
- Also allows for
  - Content negotiation (eg. Language, media type)
  - Authentication (password-style)

HTTP Versions

- HTTP 0.9 (1991) — The first documented version
- HTTP 1.0 (RFC 1945, 1996)
- HTTP 1.1 (RFC 2068, 1997)
- HTTP 1.1 (RFC 2616, 1999) — with improvements and updates (most popular old version)
- HTTP/2 (RFC 7540, 2015) — server push, pipelining of requests, multiplexing multiple requests over a single TCP connection, and better performance
- HTTP/3 (RFC 9114, 2022) — uses QUIC (a multiplexed transport protocol built on UDP)
  - used by **29.2%** of all the websites (W3Techs, Apr 2024)

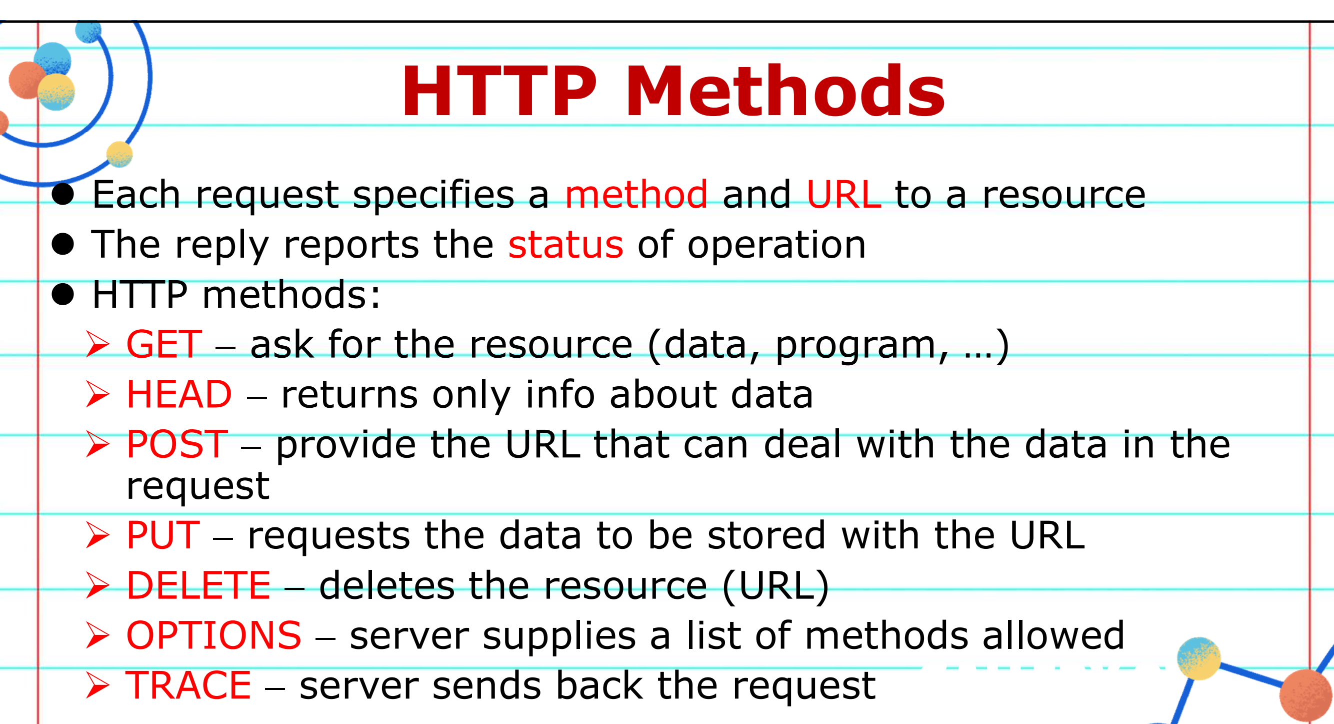
HTTP Interaction

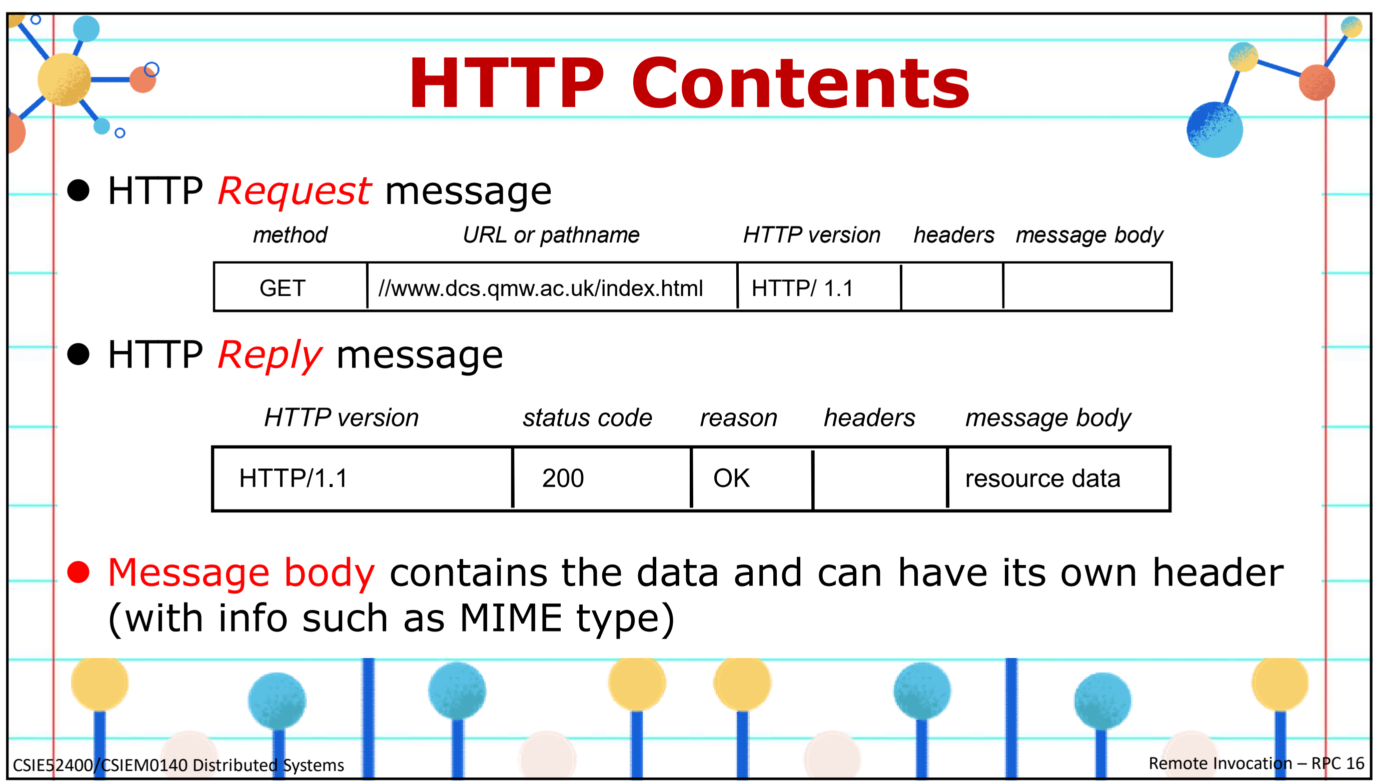


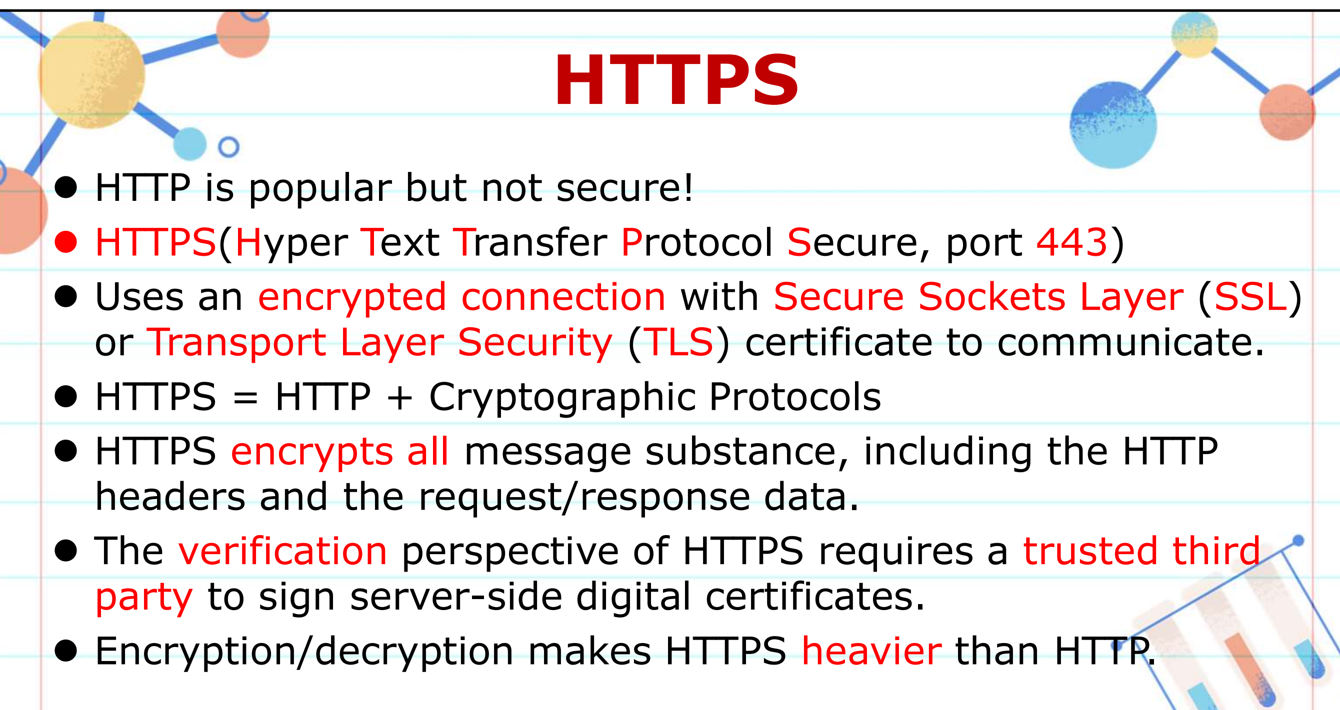
- Client-server interaction
  - Client initiates an attempt
  - Server accepts a connection on default port
  - Client sends a request message
  - Server sends a reply message
  - Connection close
- Persist connection allow a series of exchanges
- Request and replies are marshalled into text messages
- Resources can be byte sequences and compressed

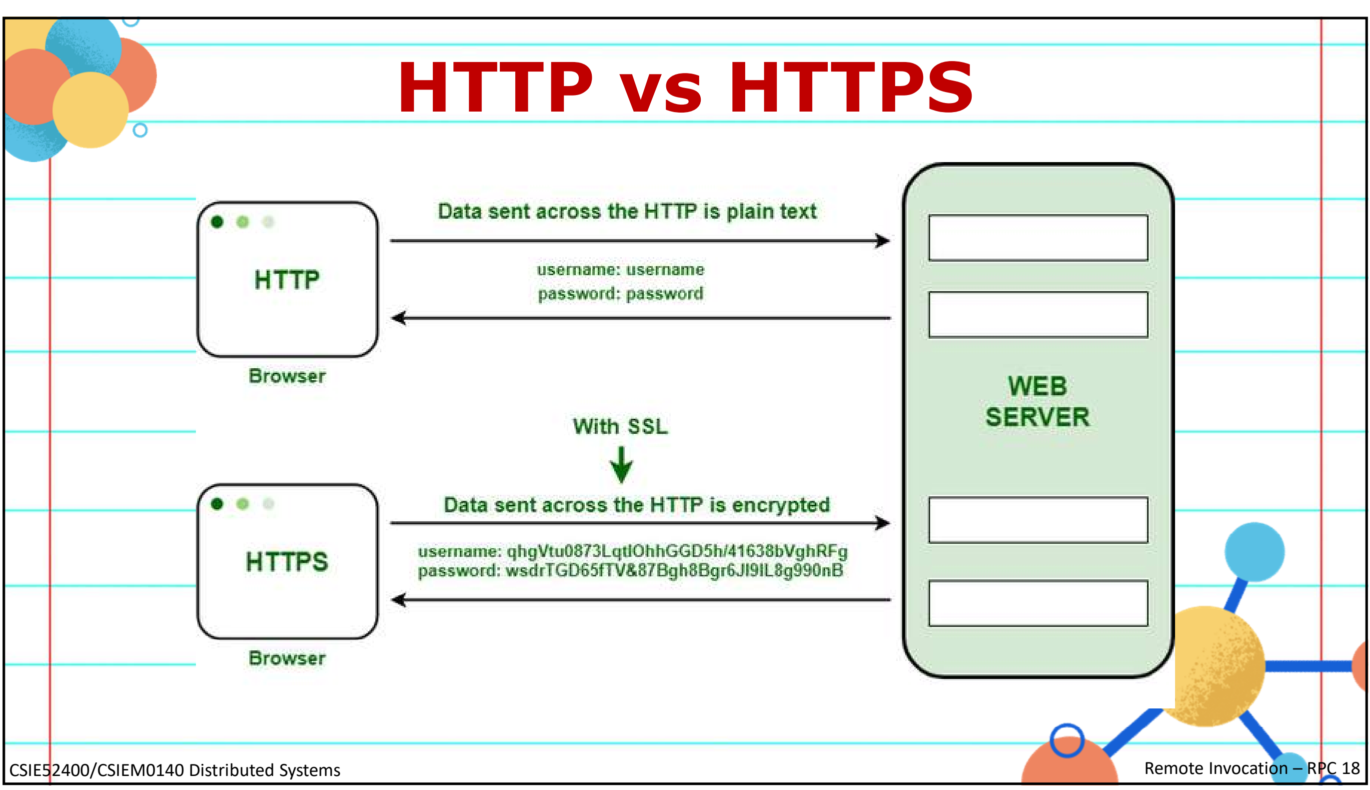
HTTP Data

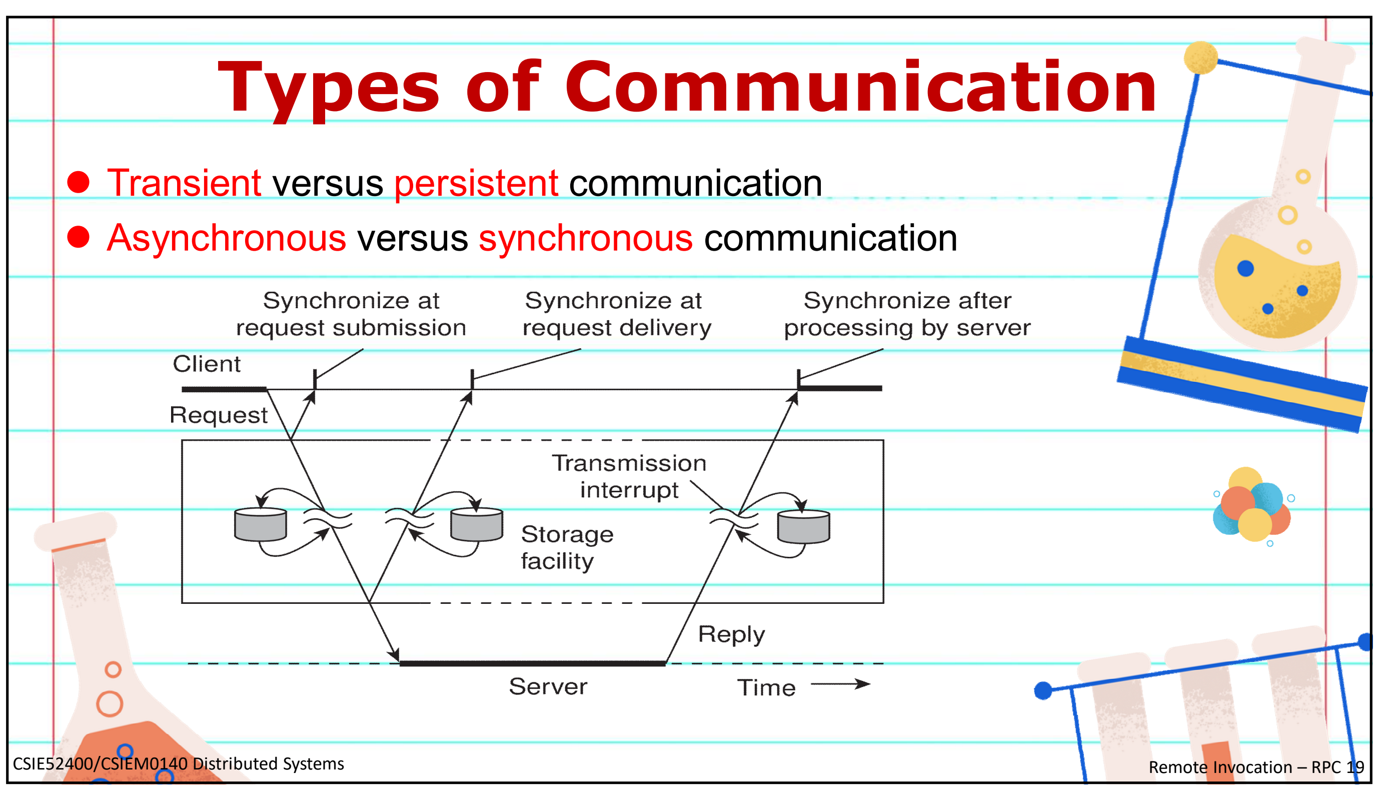
- Data are supplied as MIME-like structures
- Multipurpose Internet Mail Extensions (MIME)
- Each is prefixed with MIME type
- A MIME type specifies a type and subtype, eg. text/plain, text/html, image/gif, image/jpeg, …
- Clients can specify the MIME types to accept

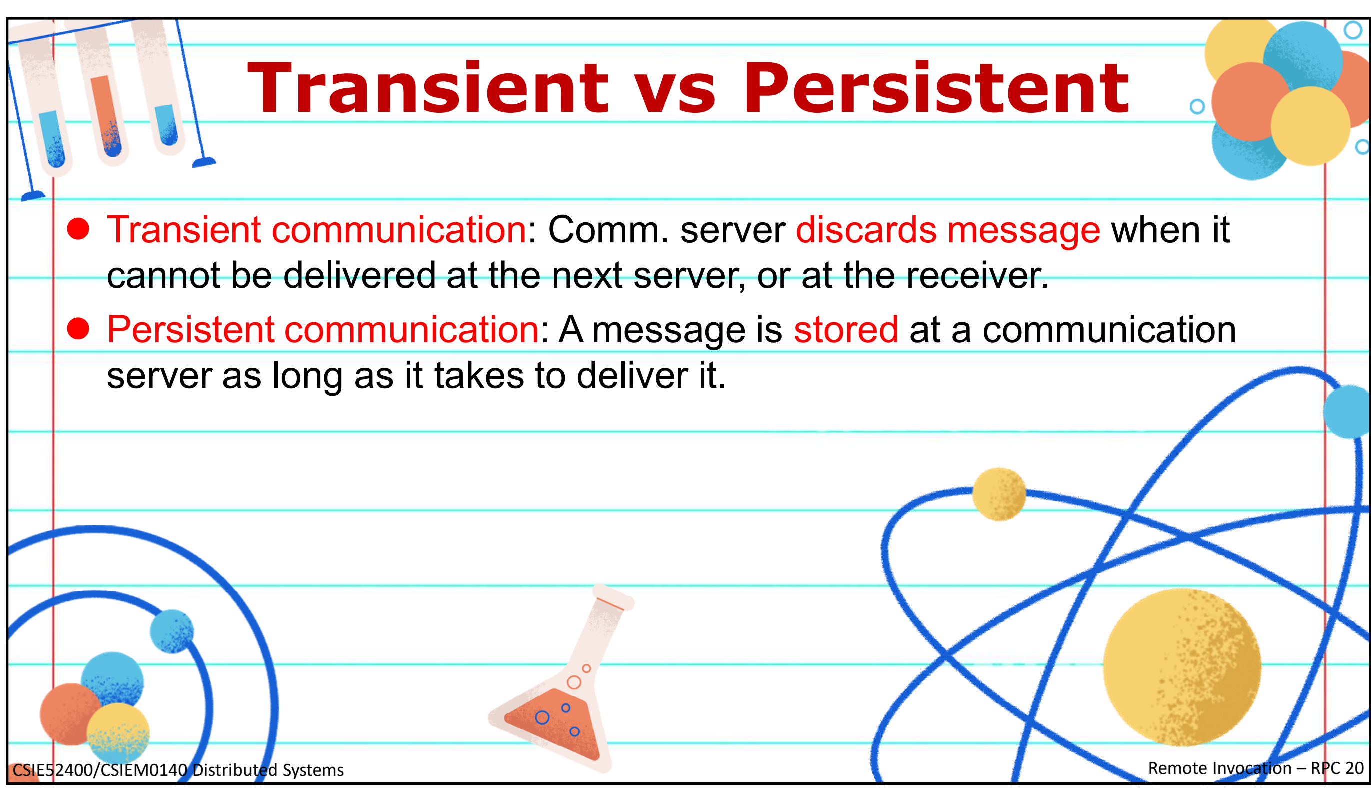

HTTP Methods
Each request specifies a method and URL to a resource
The reply reports the status of operation
HTTP methods:
GET – ask for the resource (data, program, …)
HEAD – returns only info about data
POST – provide the URL that can deal with the data in the request
PUT – requests the data to be stored with the URL
DELETE – deletes the resource (URL)
OPTIONS – server supplies a list of methods allowed
TRACE – server sends back the request
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 15

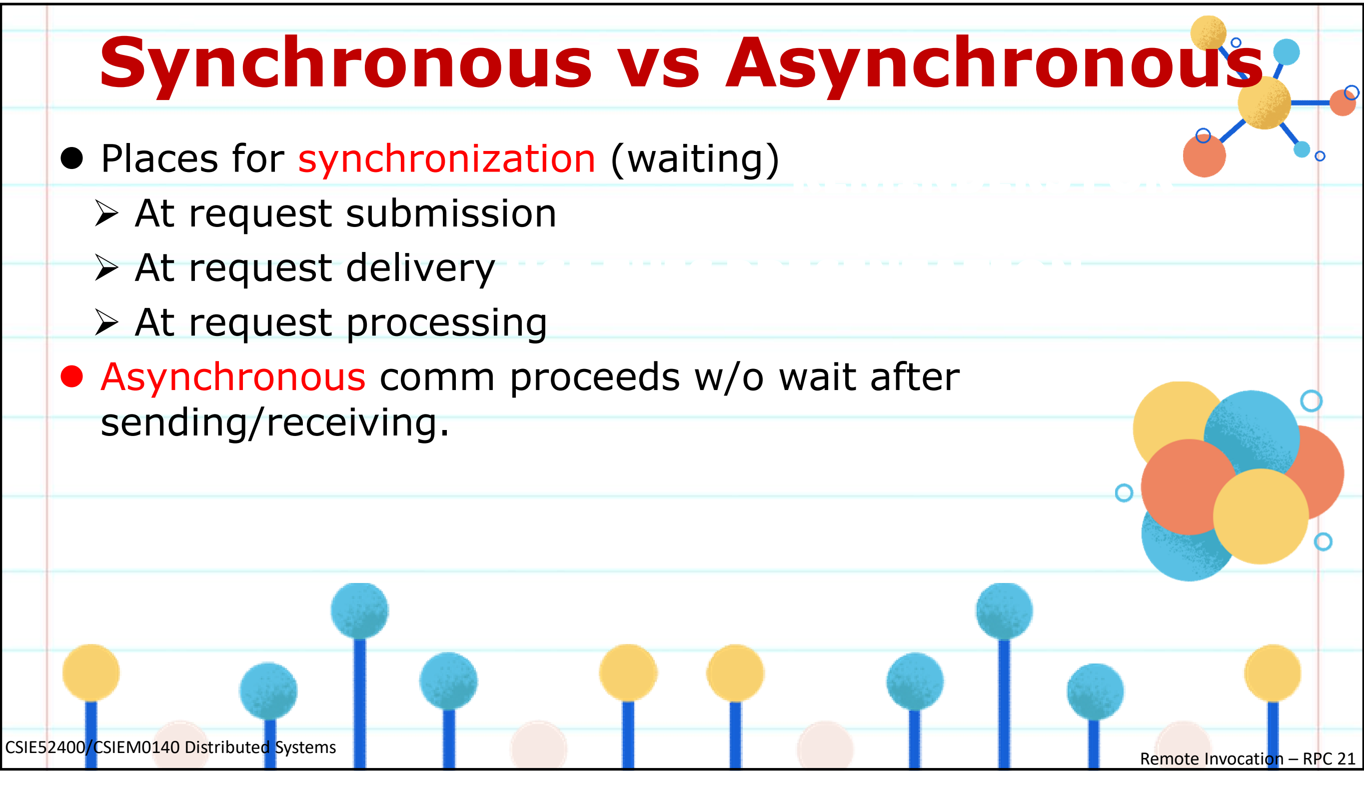

HTTP Contents
HTTP Request message
method
URL or pathname
HTTP version
headers
message body
GET
//www.dcs.qmw.ac.uk/index.html
HTTP/ 1.1
HTTP Reply message
HTTP version
status code
reason
headers
message body
HTTP/1.1
200
OK
resource data
Message body contains the data and can have its own header (with info such as MIME type)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 16


HTTPS
HTTP is popular but not secure!
HTTPS(Hyper Text Transfer Protocol Secure, port 443)
Uses an encrypted connection with Secure Sockets Layer (SSL) or Transport Layer Security (TLS) certificate to communicate.
HTTPS = HTTP + Cryptographic Protocols
HTTPS encrypts all message substance, including the HTTP headers and the request/response data.
The verification perspective of HTTPS requires a trusted third party to sign server-side digital certificates.
Encryption/decryption makes HTTPS heavier than HTTP.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 17


HTTP vs HTTPS
HTTP
Browser
Data sent across the HTTP is plain text
username: username
password: password
With SSL
HTTPS
Browser
Data sent across the HTTP is encrypted
username: qhgVtu0873LqtlOhhGGD5h/41638bVghRFg
password: wsdrTGD65fTV&87Bgh8Bgr6JI9IL8g990nB
WEB
SERVER
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 18

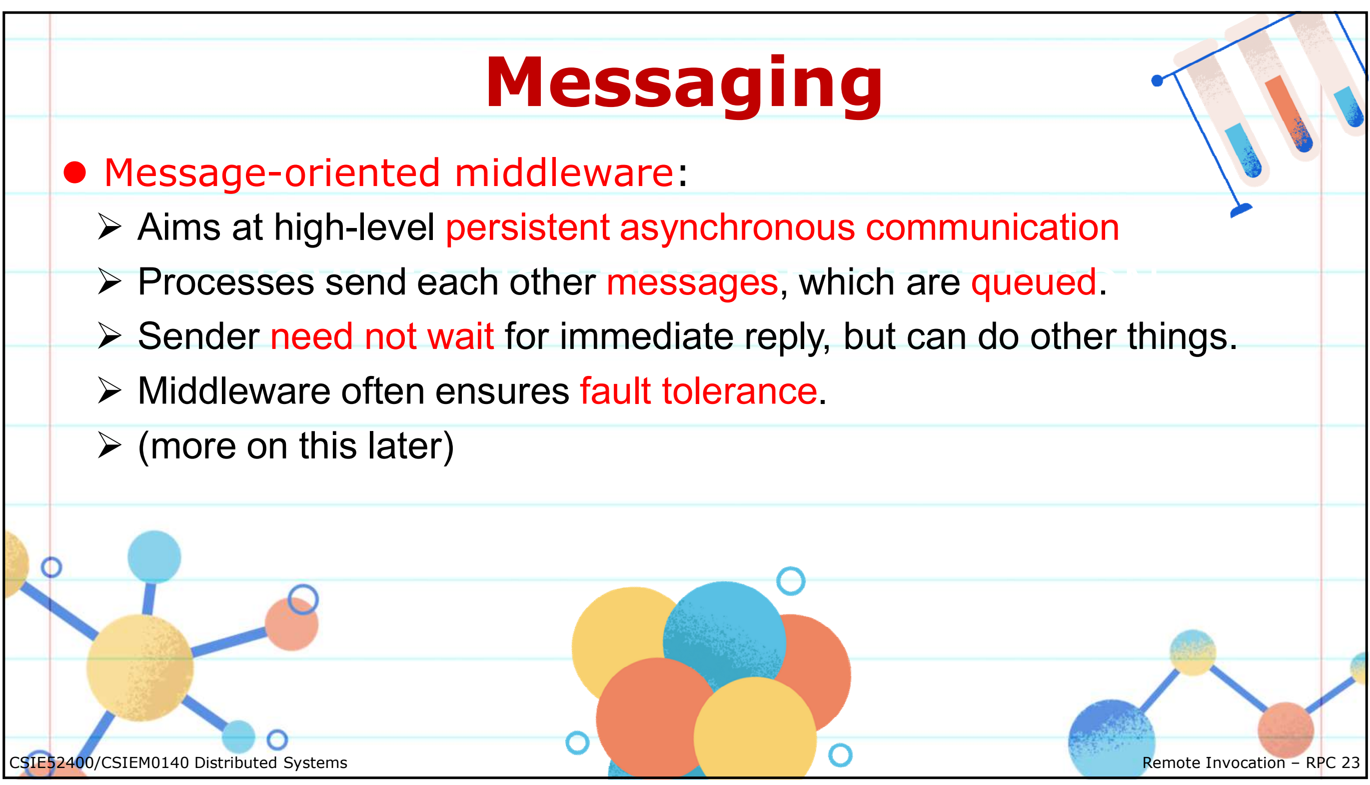

Types of Communication
Transient versus persistent communication
Asynchronous versus synchronous communication
Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 19

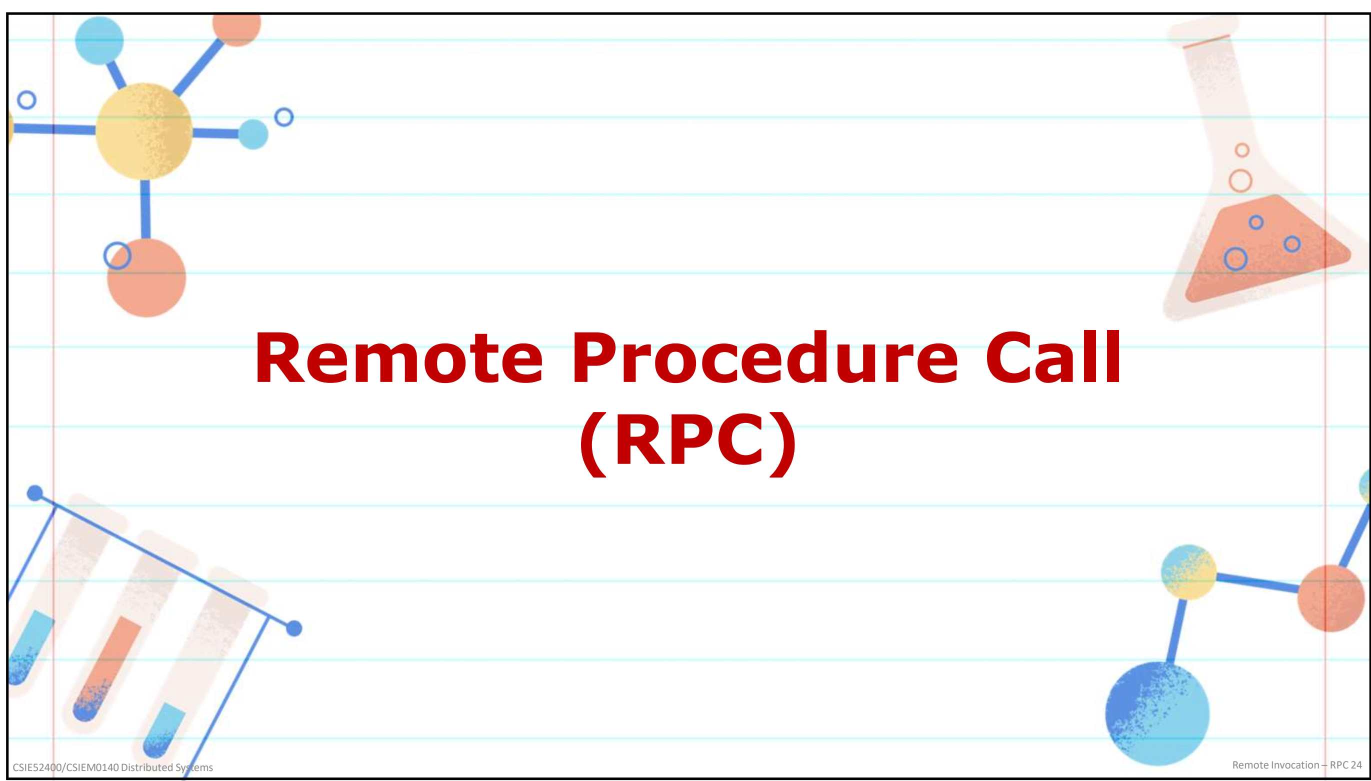

Transient vs Persistent
Transient communication: Comm. server discards message when it cannot be delivered at the next server, or at the receiver.
Persistent communication: A message is stored at a communication server as long as it takes to deliver it.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 20



Client/Server Comm

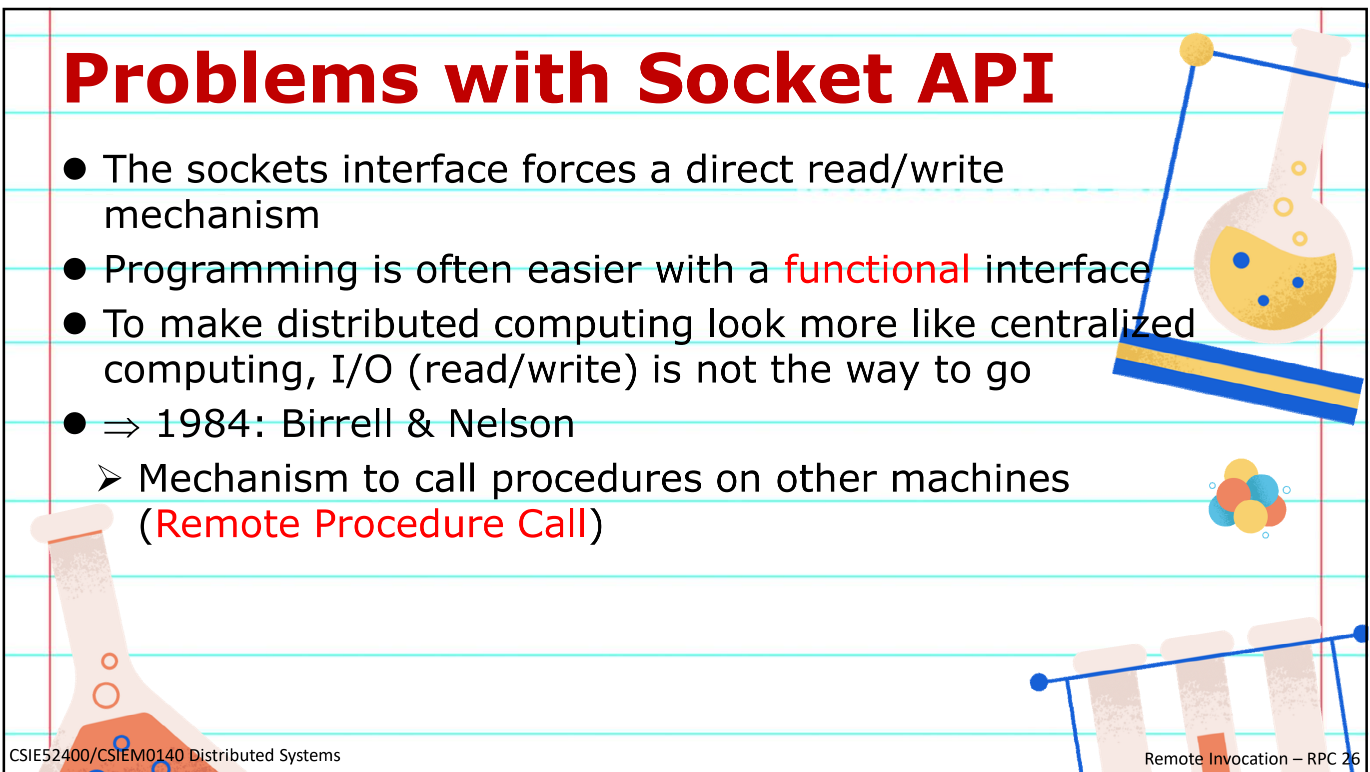
- Client/Server computing is generally based on a model of transient synchronous communication:
  - Client and server have to be active at the time of communication
  - Client issues request and blocks until it receives reply
  - Server essentially waits only for incoming requests, and subsequently processes them
- **Drawbacks** of synchronous communication:
  - Client cannot do any other work while waiting for reply
  - Failures have to be handled immediately: the client is waiting
  - The model may simply not be appropriate (eg. mail, news)


Messaging
● Message-oriented middleware:
➢ Aims at high-level persistent asynchronous communication
➢ Processes send each other messages, which are queued.
➢ Sender need not wait for immediate reply, but can do other things.
➢ Middleware often ensures fault tolerance.
➢ (more on this later)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 23


Remote Procedure Call
(RPC)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 24

Recap: Socket-based Comm

- Socket API: all we get from the OS to access the network
- Socket = distinct end-to-end comm channels
- Read/write model
  - Send a bunch of bytes
  - Read a bunch of bytes
  - Send a bunch of bytes
  - Read a bunch of bytes
  - …
- Application implements its protocol
- Line-oriented, text-based protocols common
  - Not efficient but easy to debug & use



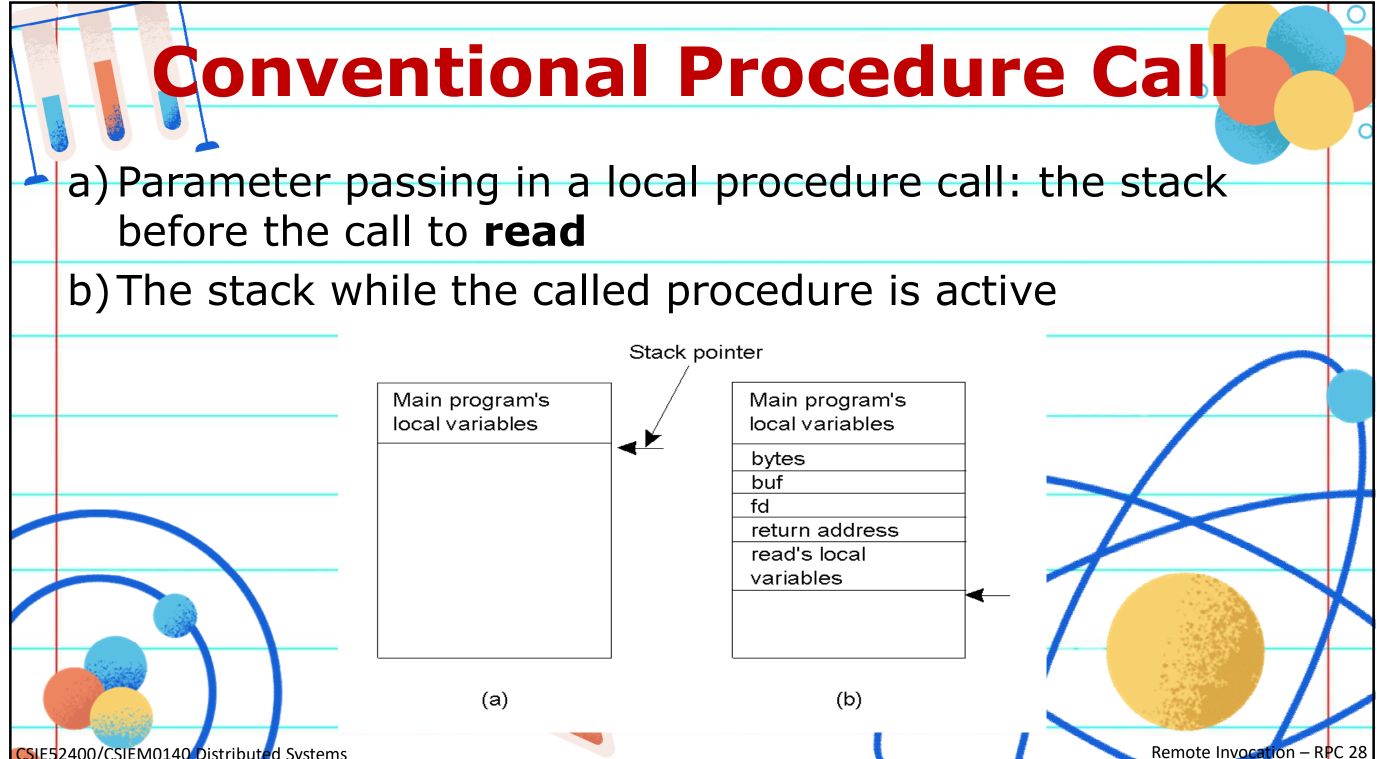
Conventional Procedure Call

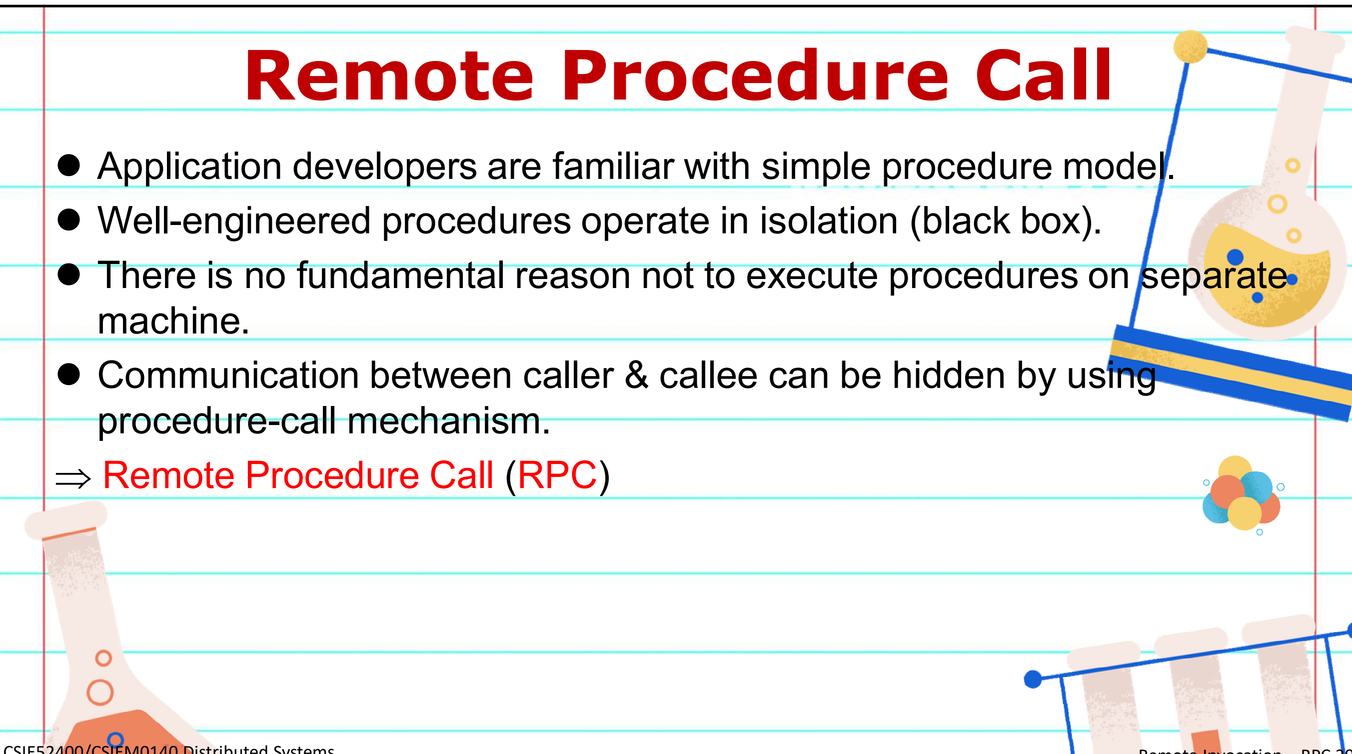
- You write: x = f(a, "test", 5);
- The compiler parses this and generates code to:
  1. Push the value 5 on the stack
  2. Push the address of the string "test" on the stack
  3. Push the current value of a on the stack
  4. Generate a call to the function f
- In compiling f, the compiler generates code to:
  1. Push registers that will be clobbered on the stack to save the values
  2. Adjust the stack to make room for local and temporary variables
  3. Before a return, unadjust the stack, put the return data in a register, and issue a return instruction

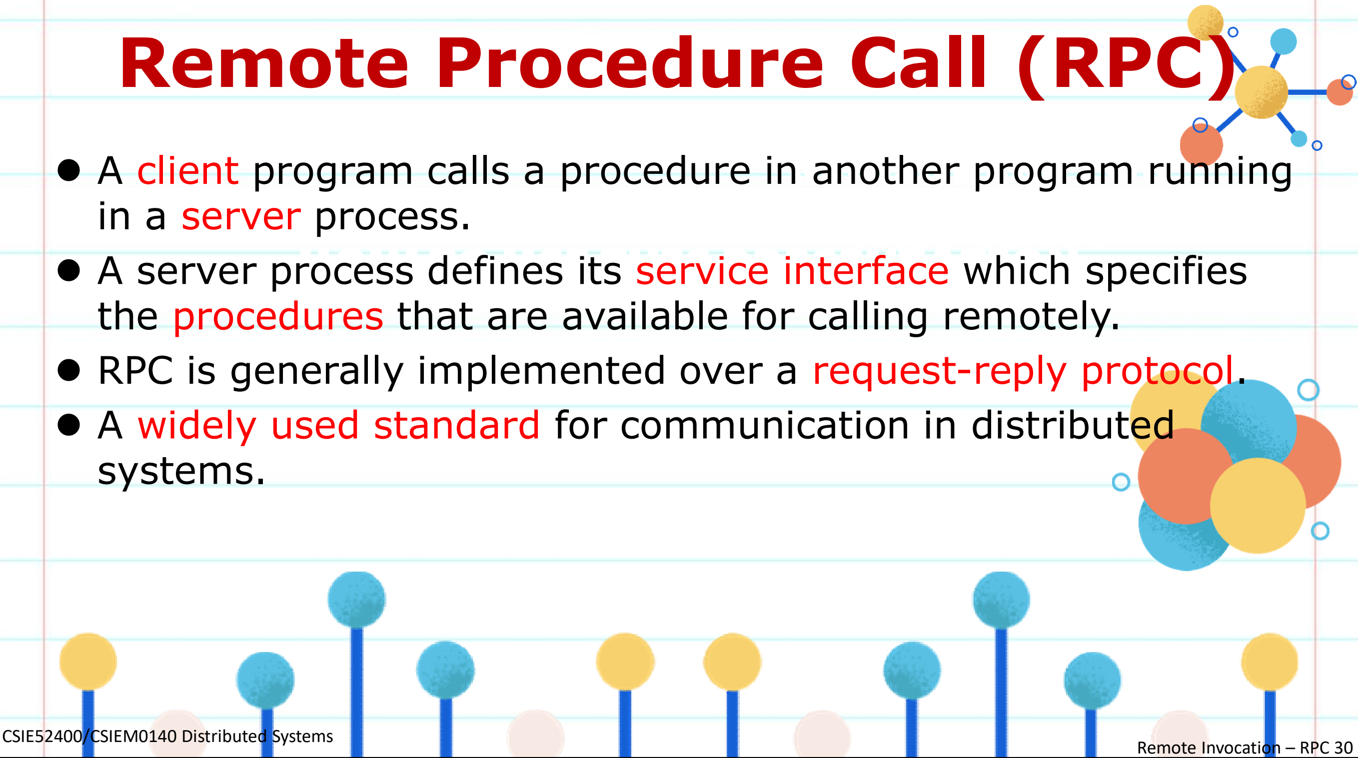
Conventional Procedure Call

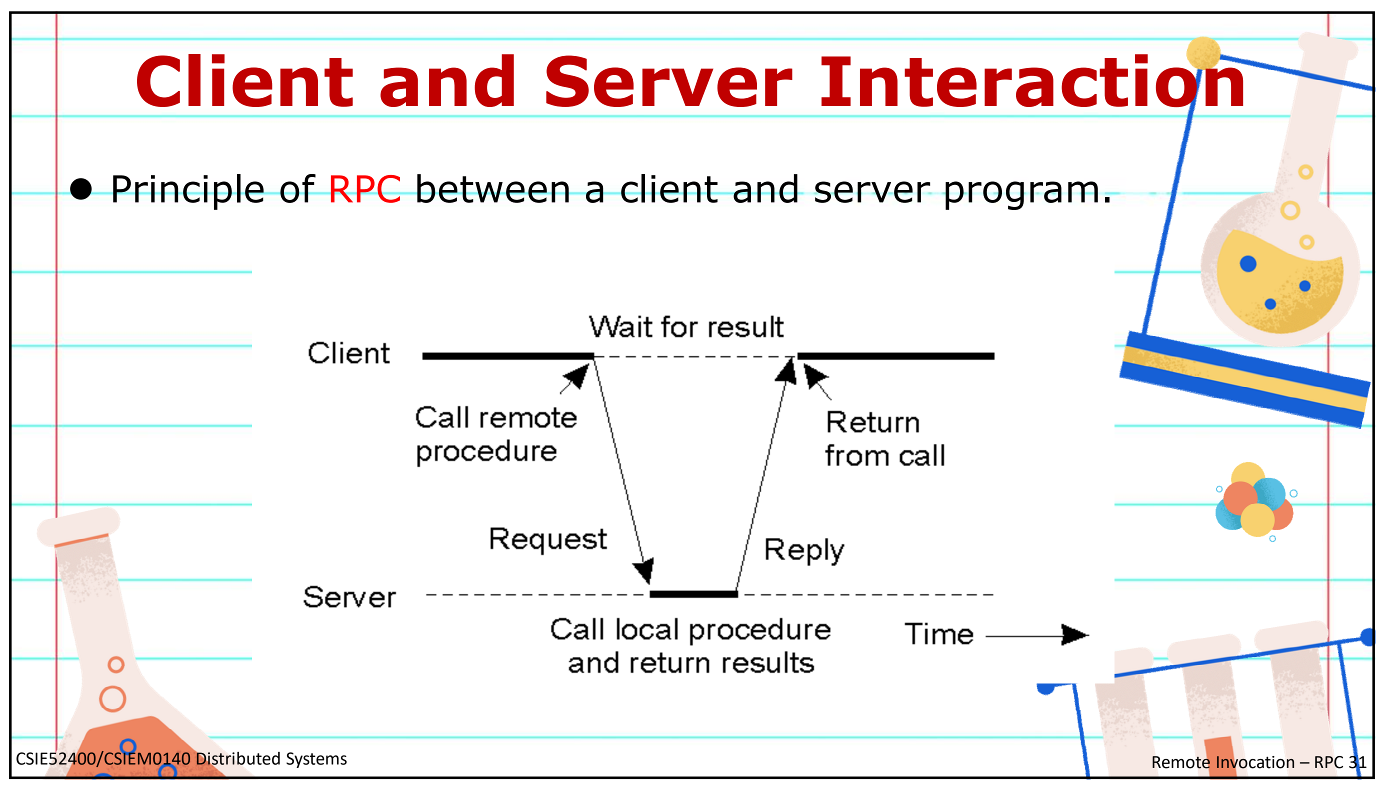
a) Parameter passing in a local procedure call: the stack before the call to **read**

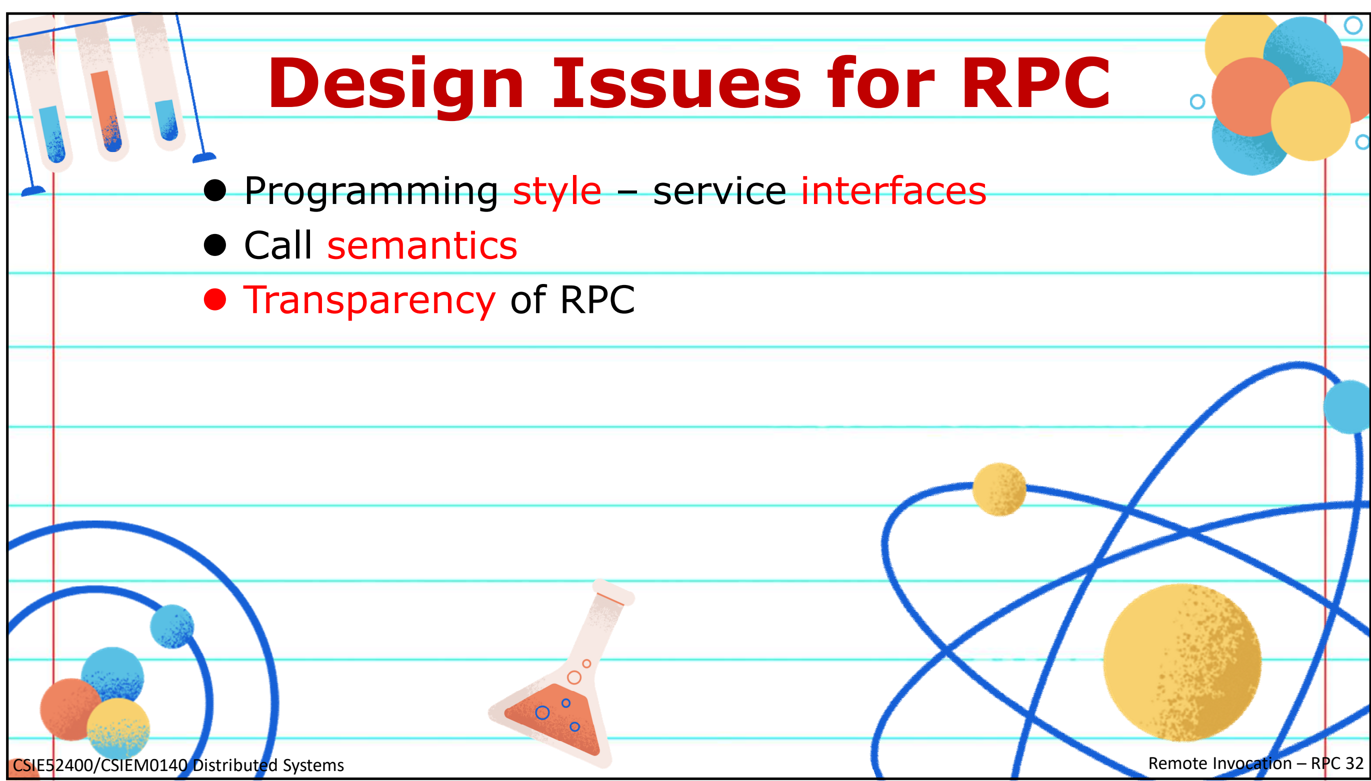
b) The stack while the called procedure is active

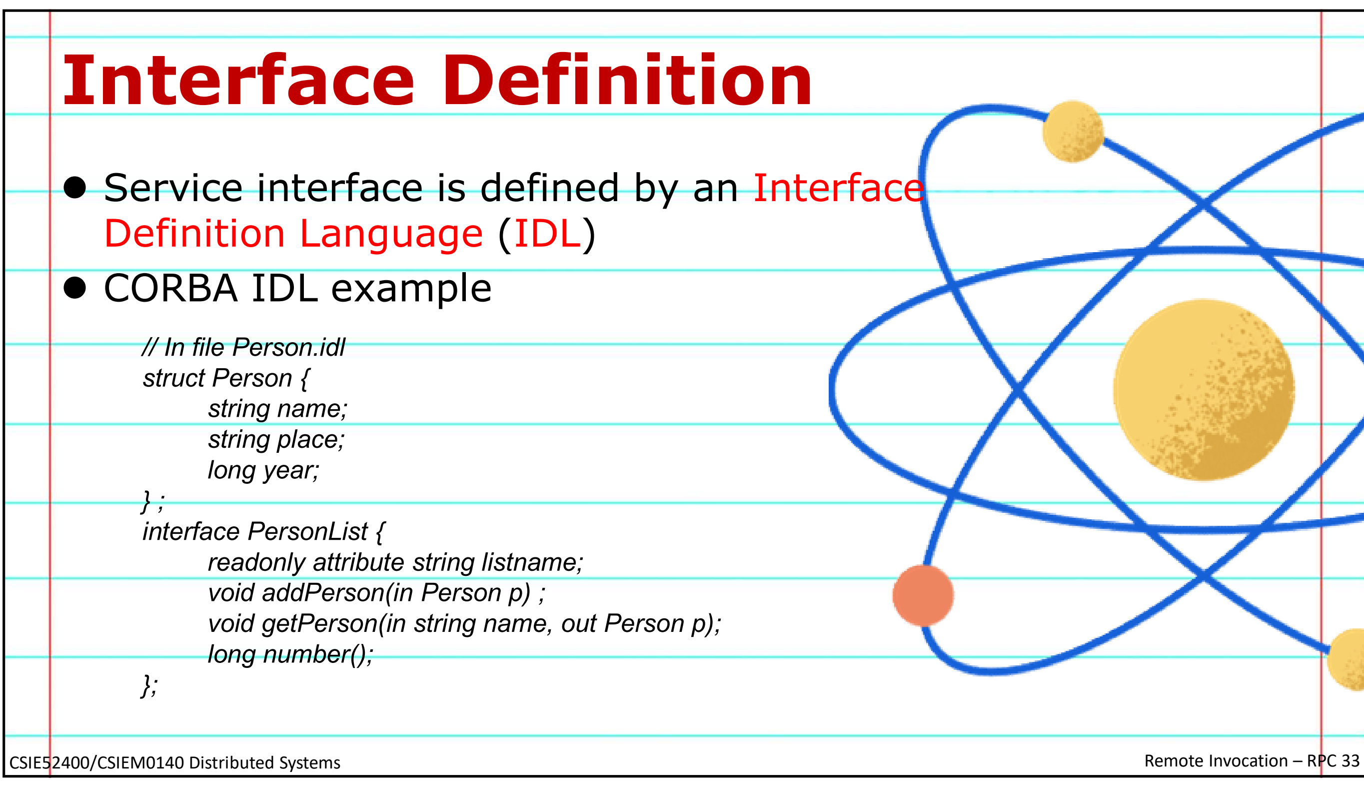


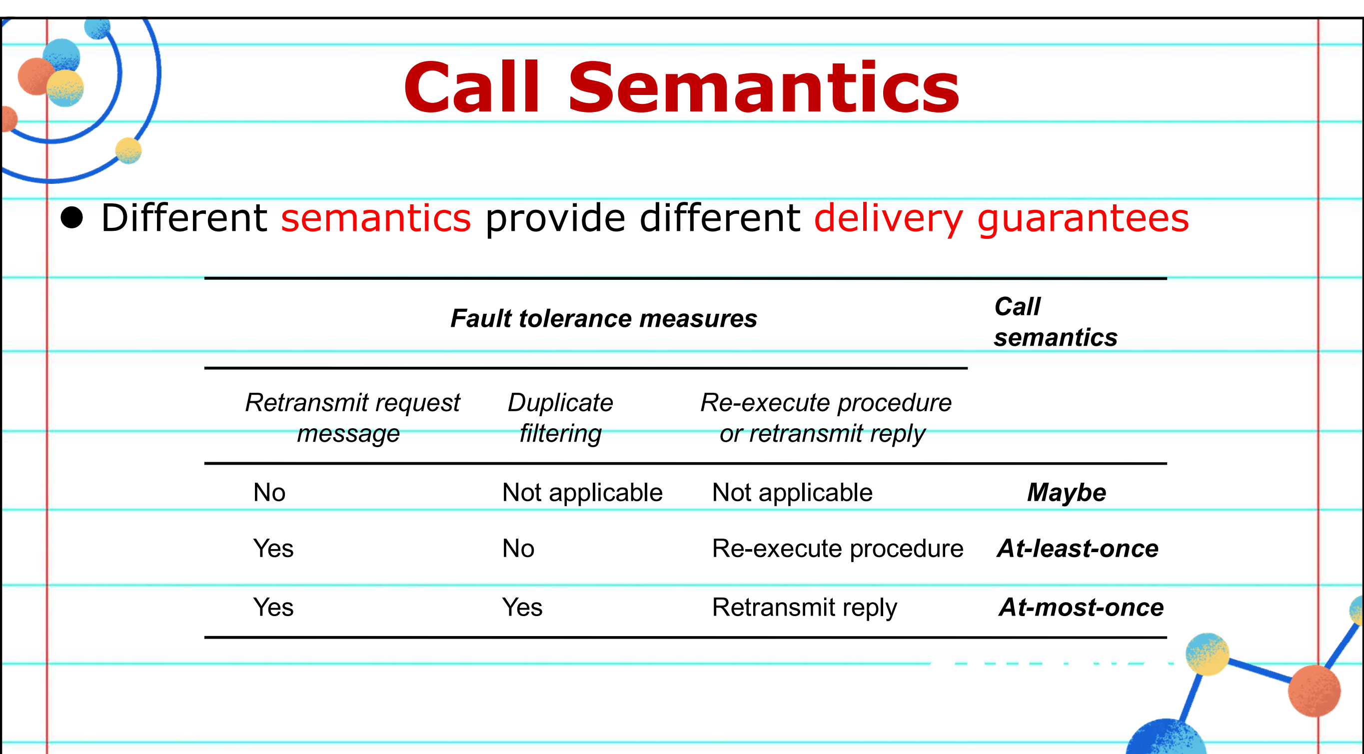

Remote Procedure Call
Application developers are familiar with simple procedure model.
Well-engineered procedures operate in isolation (black box).
There is no fundamental reason not to execute procedures on separate machine.
Communication between caller & callee can be hidden by using procedure-call mechanism.
⇒ Remote Procedure Call (RPC)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 29

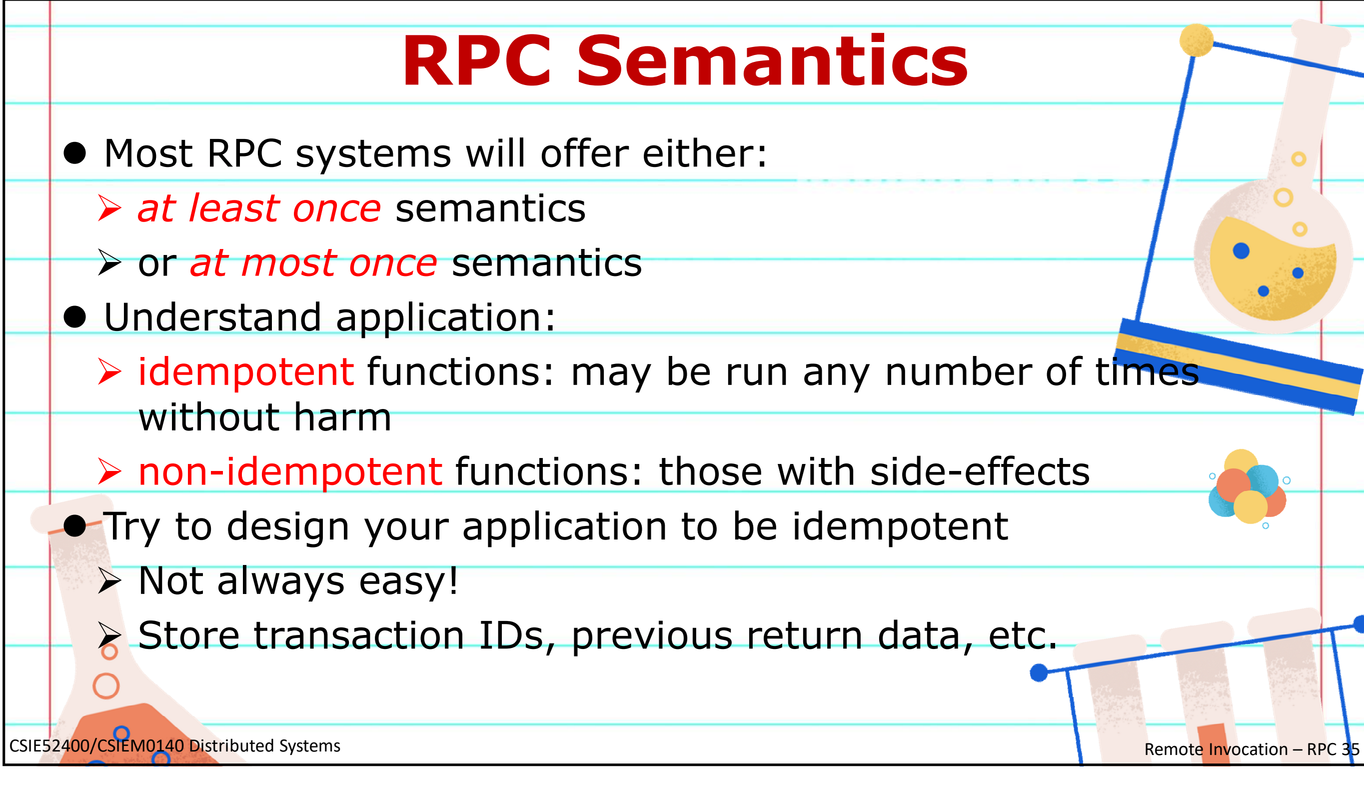

Remote Procedure Call (RPC)
A client program calls a procedure in another program running in a server process.
A server process defines its service interface which specifies the procedures that are available for calling remotely.
RPC is generally implemented over a request-reply protocol.
A widely used standard for communication in distributed systems.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 30

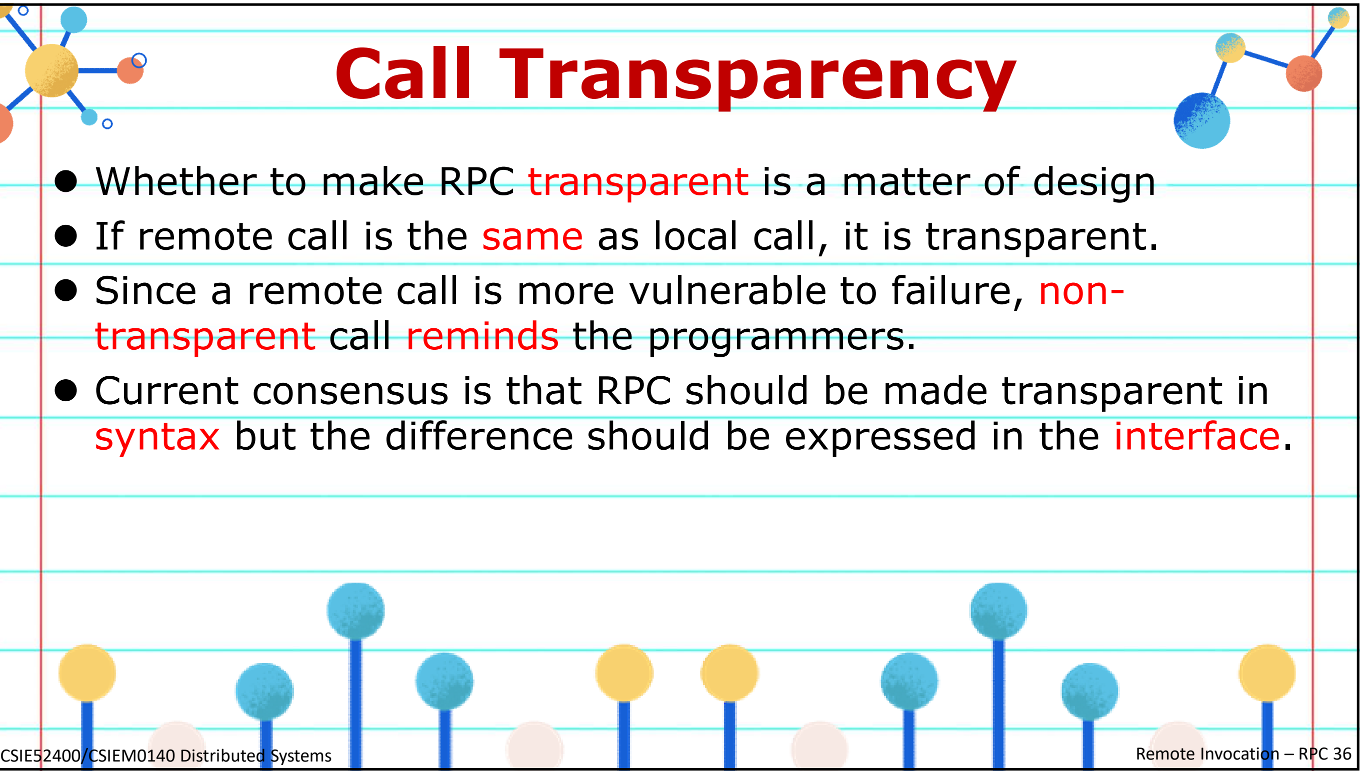

Client and Server Interaction
Principle of RPC between a client and server program.
Client
Wait for result
Call remote procedure
Return from call
Request
Reply
Server
Call local procedure and return results
Time
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 31

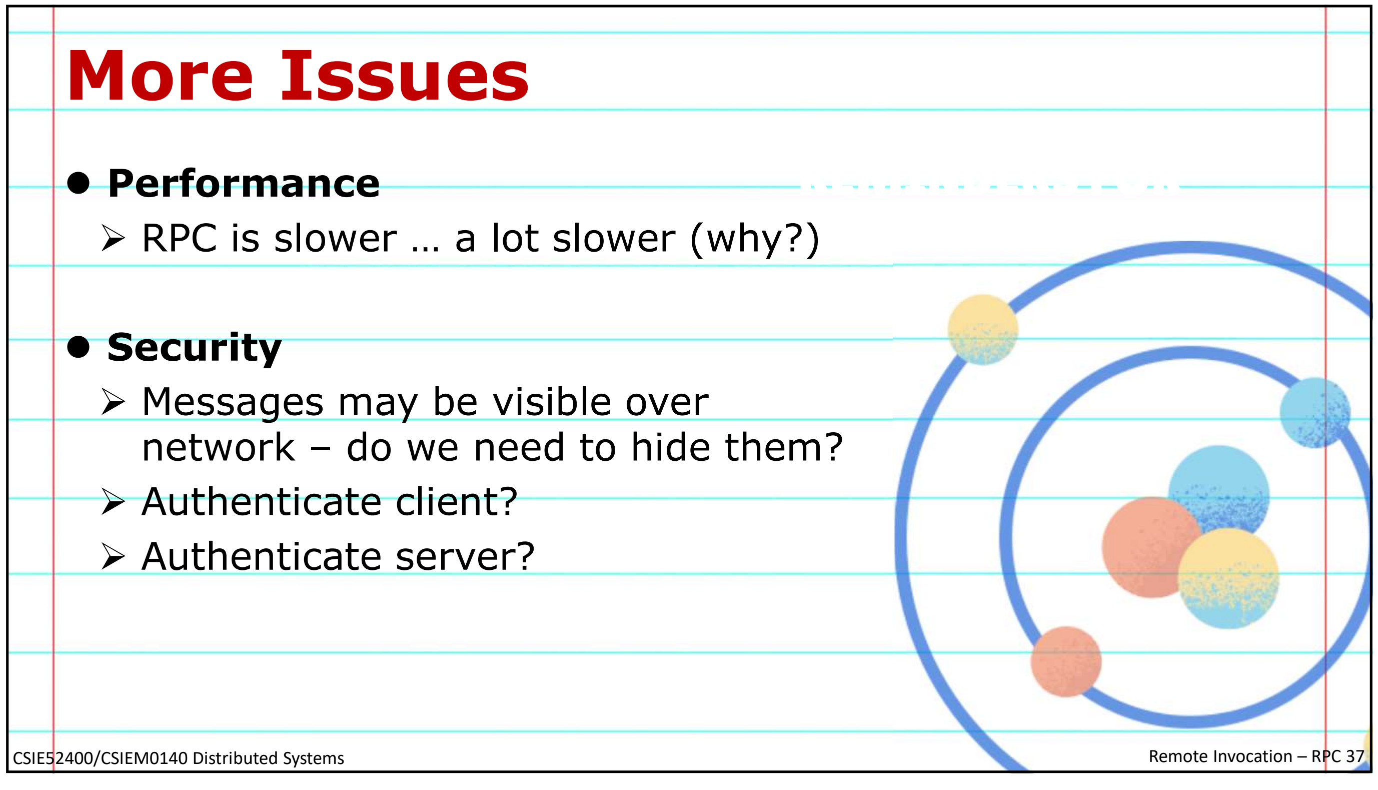

Design Issues for RPC
Programming style – service interfaces
Call semantics
Transparency of RPC
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 32

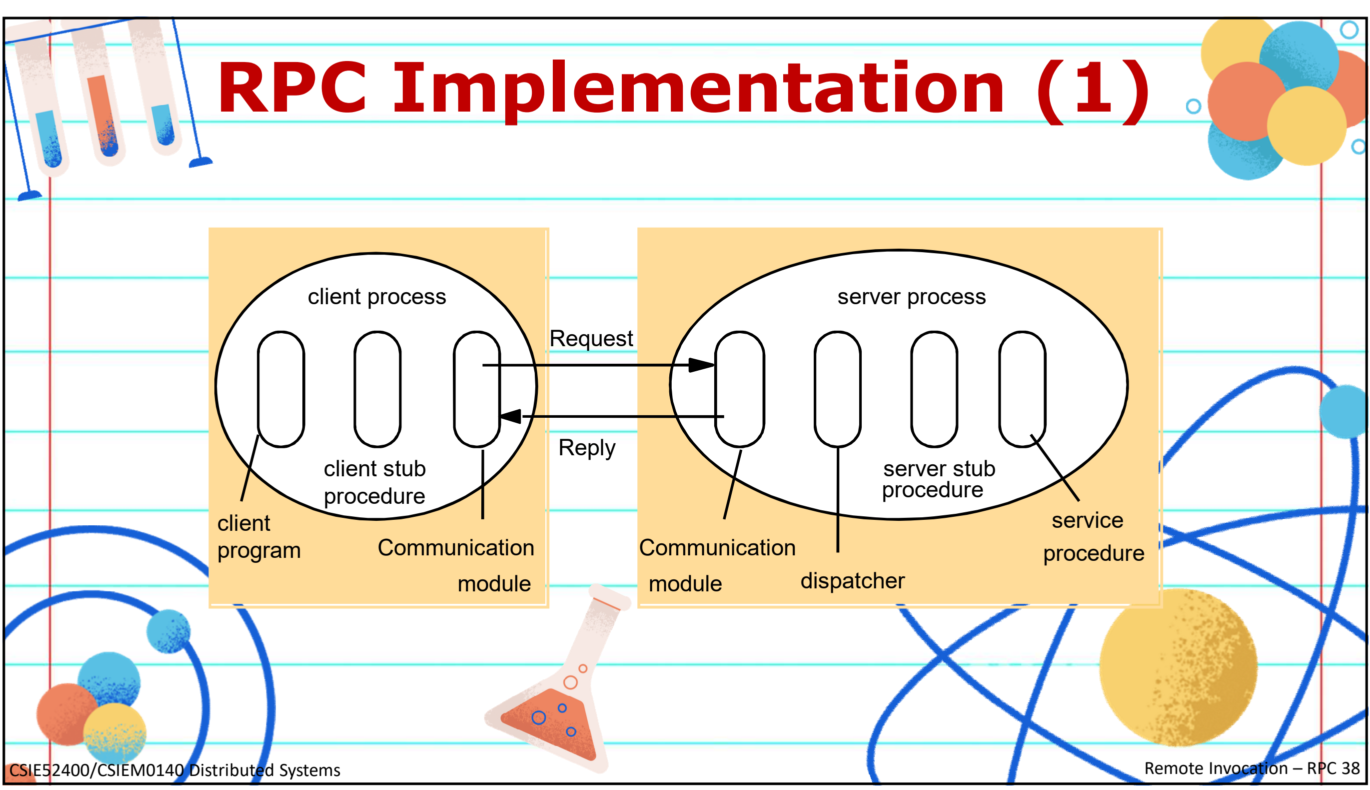

Interface Definition
Service interface is defined by an Interface Definition Language (IDL)
CORBA IDL example
// In file Person.idl
struct Person {
string name;
string place;
long year;
} ;
interface PersonList {
readonly attribute string listname;
void addPerson(in Person p) ;
void getPerson(in string name, out Person p);
long number();
};
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 33

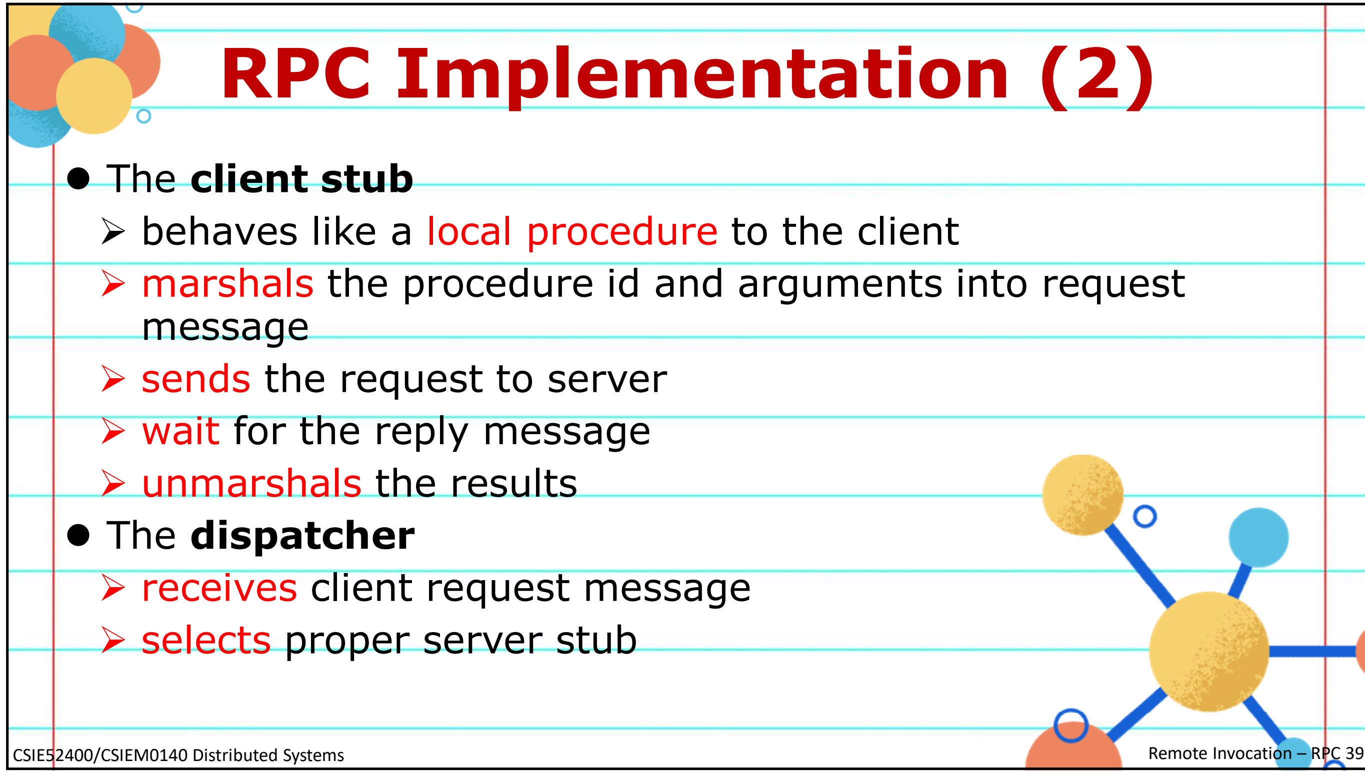

Call Semantics
Different semantics provide different delivery guarantees
Fault tolerance measures
Call semantics
Retransmit request message
Duplicate filtering
Re-execute procedure or retransmit reply
No Not applicable Not applicable Maybe
Yes No Re-execute procedure At-least-once
Yes Yes Retransmit reply At-most-once
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 34

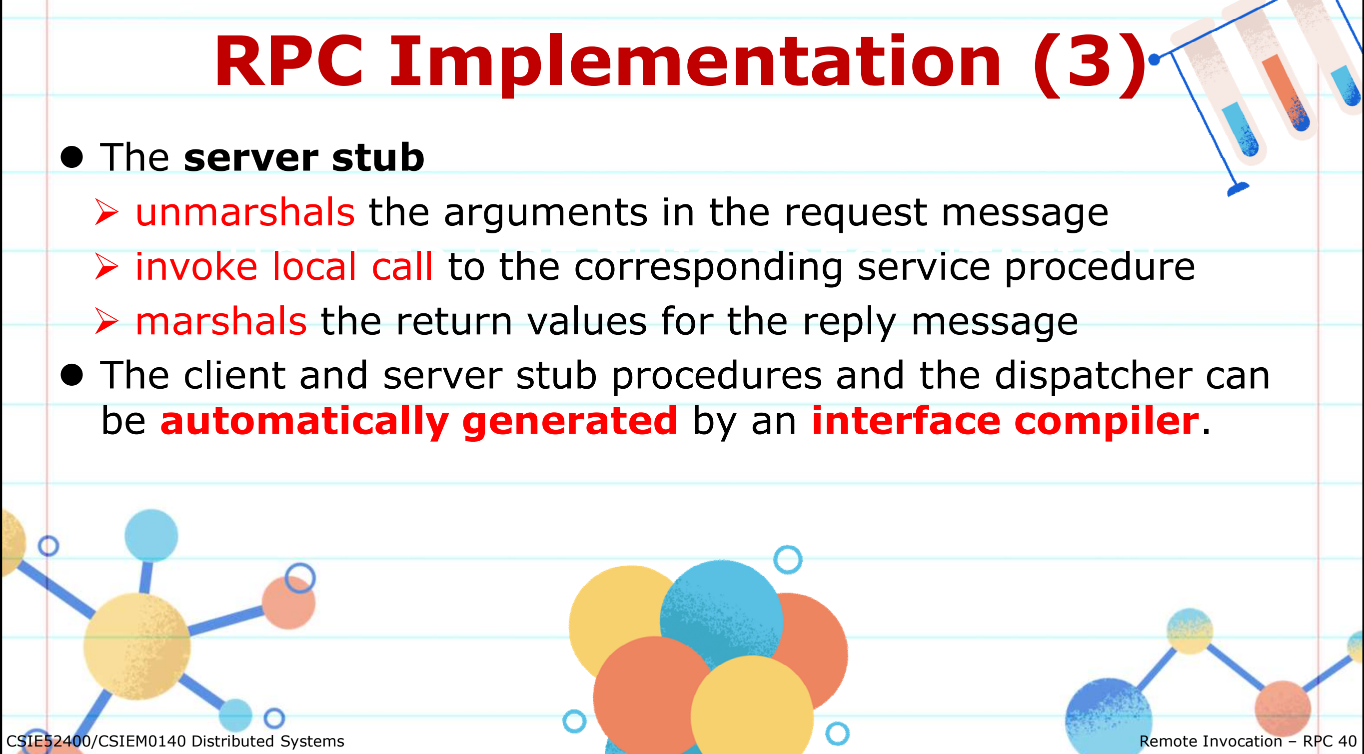

RPC Semantics
● Most RPC systems will offer either:
➢ at least once semantics
➢ or at most once semantics
● Understand application:
➢ idempotent functions: may be run any number of times without harm
➢ non-idempotent functions: those with side-effects
● Try to design your application to be idempotent
➢ Not always easy!
➢ Store transaction IDs, previous return data, etc.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 35

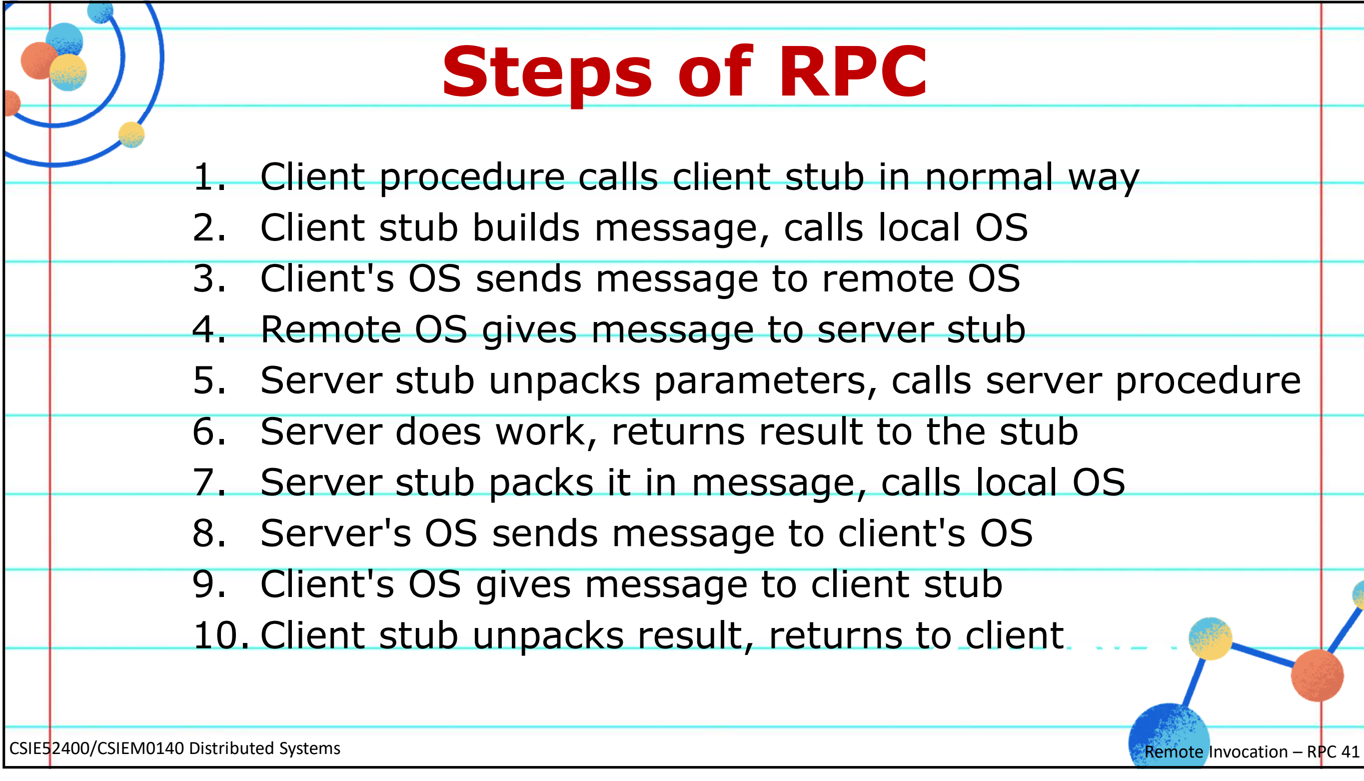

Call Transparency
● Whether to make RPC transparent is a matter of design
● If remote call is the same as local call, it is transparent.
● Since a remote call is more vulnerable to failure, non-transparent call reminds the programmers.
● Current consensus is that RPC should be made transparent in syntax but the difference should be expressed in the interface.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 36

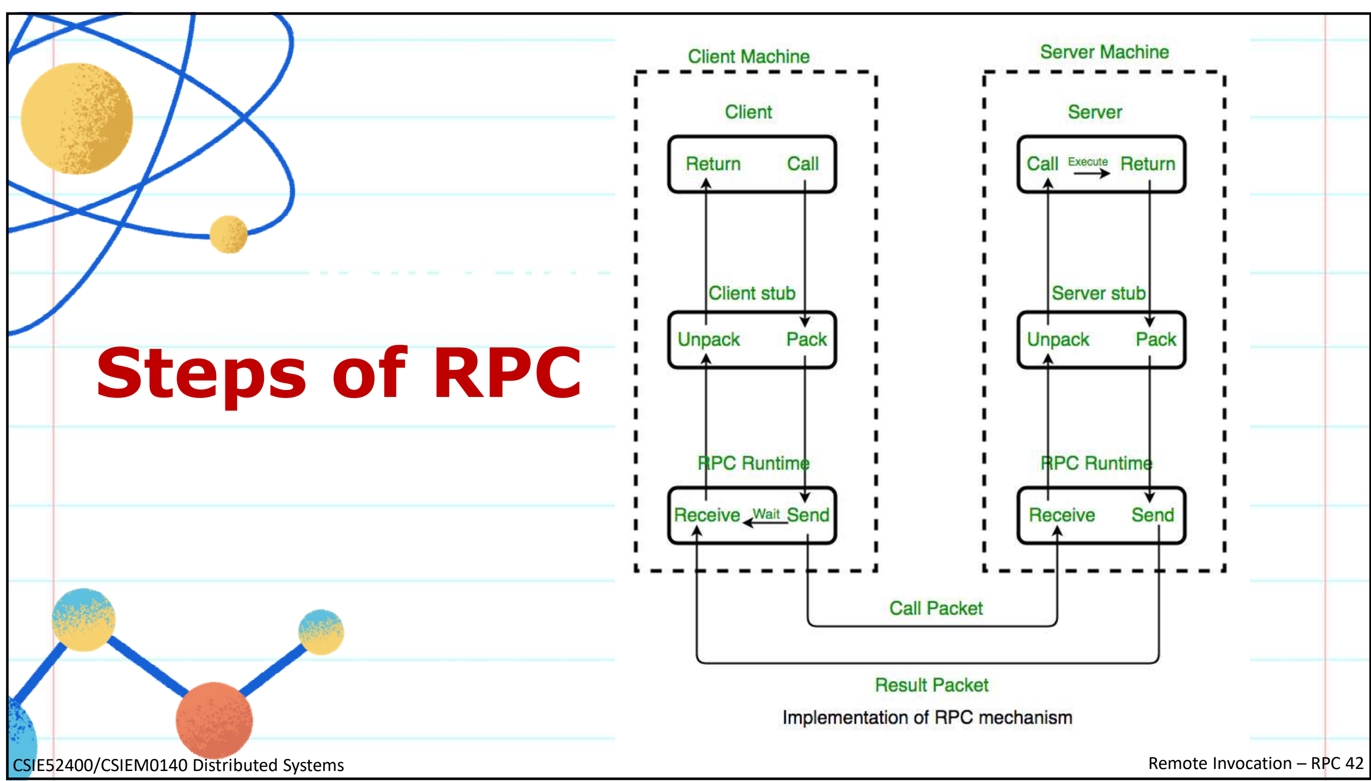

More Issues
Performance
RPC is slower … a lot slower (why?)
Security
Messages may be visible over network – do we need to hide them?
Authenticate client?
Authenticate server?
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 37

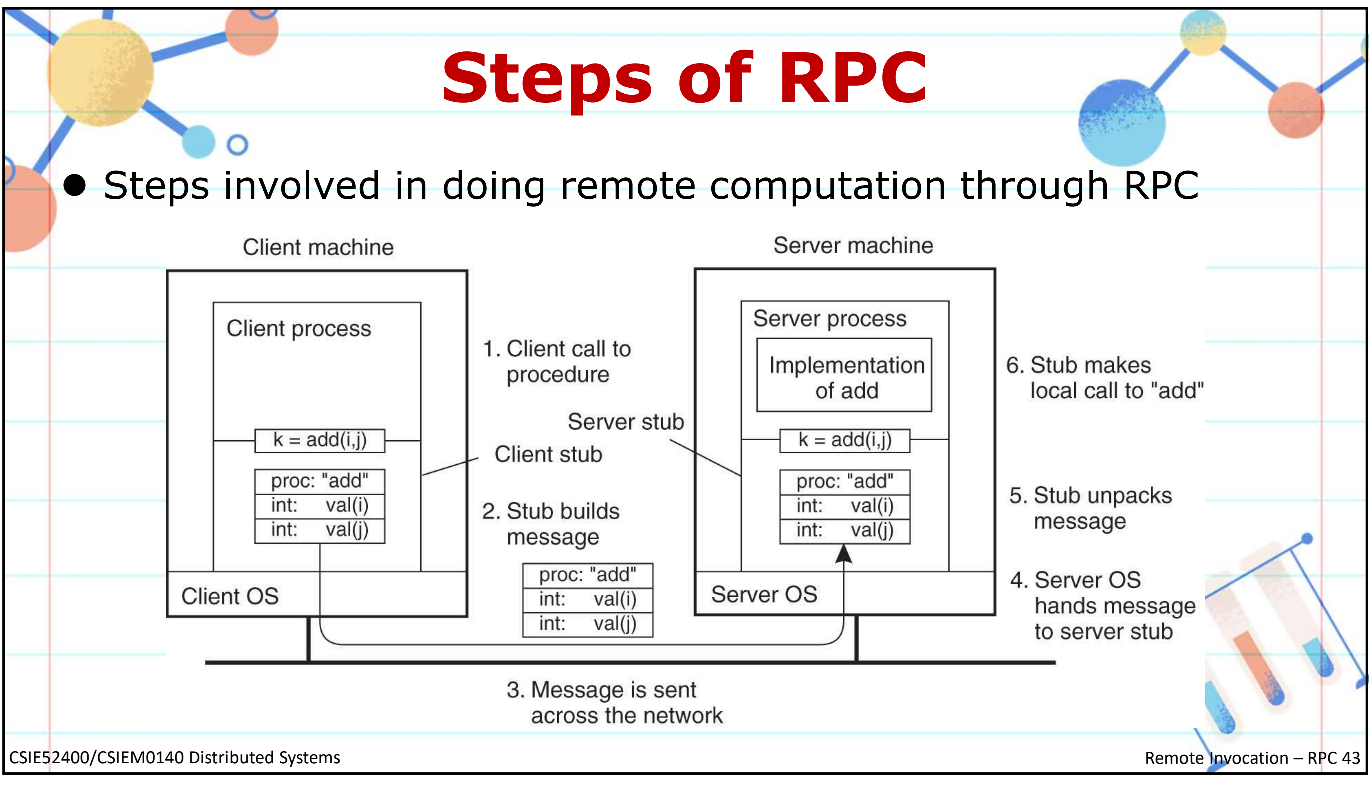

RPC Implementation (1)
client process
Request
Reply
client stub procedure
client program
Communication module
server process
server stub procedure
service procedure
Communication module
dispatcher
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 38

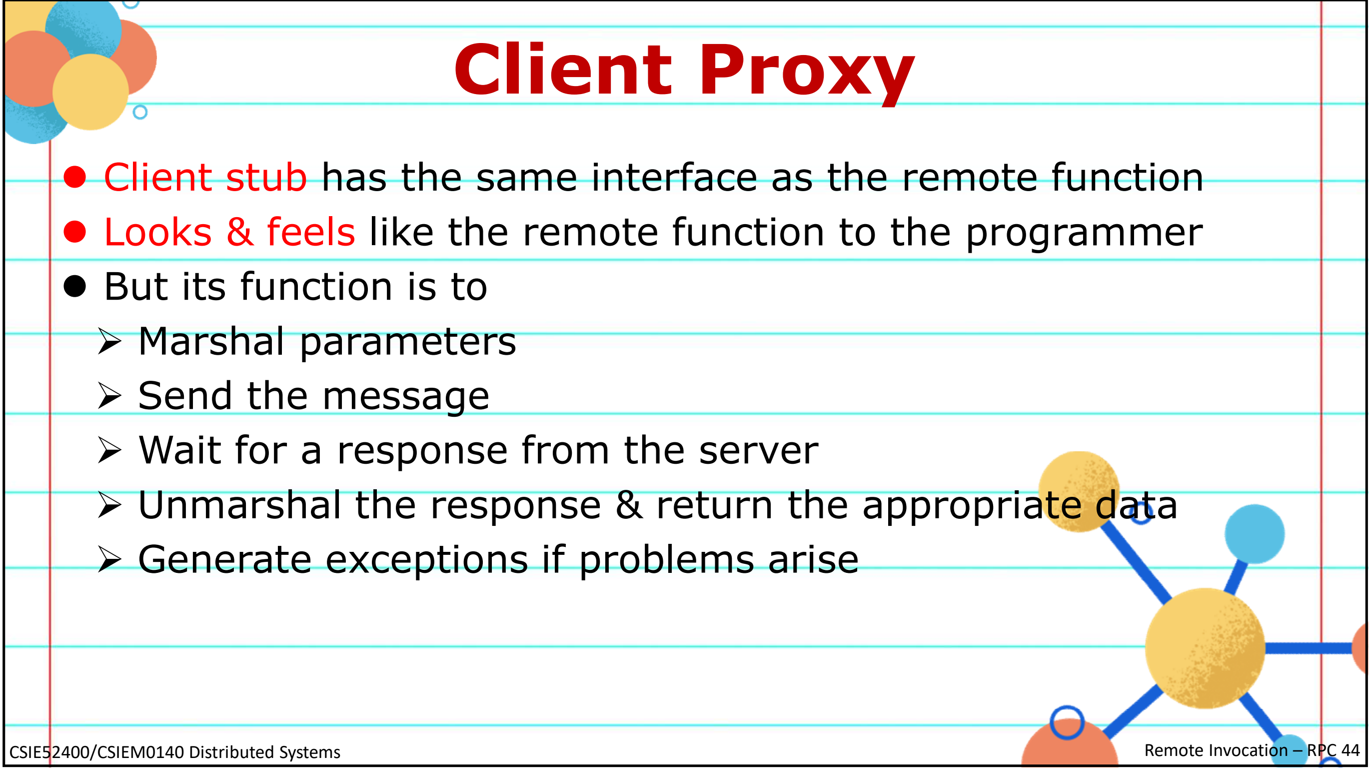

RPC Implementation (2)
The client stub
behaves like a local procedure to the client
marshals the procedure id and arguments into request message
sends the request to server
wait for the reply message
unmarshals the results
The dispatcher
receives client request message
selects proper server stub
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 39


RPC Implementation (3)
The server stub
unmarshals the arguments in the request message
invoke local call to the corresponding service procedure
marshals the return values for the reply message
The client and server stub procedures and the dispatcher can be automatically generated by an interface compiler.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 40


Steps of RPC
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 procedure
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. Client stub unpacks result, returns to client
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 41


Steps of RPC
Client Machine
Client
Return
Call
Client stub
Unpack
Pack
RPC Runtime
Receive
Wait
Send
Server Machine
Server
Call
Execute
Return
Server stub
Unpack
Pack
RPC Runtime
Receive
Send
Call Packet
Result Packet
Implementation of RPC mechanism
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 42


Steps of RPC
Steps involved in doing remote computation through RPC
Client machine
Server machine
Client process
k = add(i,j)
proc: "add"
int: val(i)
int: val(j)
Client OS
1. Client call to procedure
Server stub
Client stub
2. Stub builds message
proc: "add"
int: val(i)
int: val(j)
Server process
Implementation of add
k = add(i,j)
proc: "add"
int: val(i)
int: val(j)
Server OS
6. Stub makes local call to "add"
5. Stub unpacks message
4. Server OS hands message to server stub
3. Message is sent across the network
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 43


Client Proxy
Client stub has the same interface as the remote function
Looks & feels like the remote function to the programmer
But its function is to
Marshal parameters
Send the message
Wait for a response from the server
Unmarshal the response & return the appropriate data
Generate exceptions if problems arise
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 44

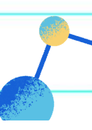
Server Stub

- Dispatcher – *the listener*
  - Receives client requests
  - Identifies appropriate function (method)
- Skeleton – *the unmarshaller & caller*
  - Unmarshals parameters
  - Calls the local server procedure
  - Marshals the response & sends it back to the dispatcher
- All this is invisible to the programmer
  - The programmer doesn't deal with any of this
  - Dispatcher + Skeleton may be integrated
    - Depends on implementation

RPC Benefits

- RPC gives us a procedure call interface
- Writing applications is simplified
  - RPC hides all network code into stub functions
  - Application programmers don't have to worry about details
    - Sockets, port numbers, byte ordering
- Where is RPC in the OSI model?
  - Layer 5: Session layer: Connection management
  - Layer 6: Presentation: Marshaling/data representation
  - Uses the transport layer (4) for communication (TCP/UDP)

RPC: Parameter Passing



- There's more than just wrapping parameters into a message
  - Client and server machines may have different data representations (think of byte ordering)
  - Wrapping a parameter means transforming a value into a sequence of bytes
  - Client and server have to agree on the same encoding
    - How are basic data values represented (integers, floats, characters)
    - How are complex data values represented (arrays, unions)
- Client and server need to properly interpret messages, transforming them into machine-dependent representations

Passing Value Parameters

a) Original message on the sender (little endian)
b) The message after receipt on the server (big endian)
c) The message after being inverted. The little numbers in boxes indicate the address of each byte

3	2	1	0
0	0	0	5
7	6	5	4
L	L	I	J

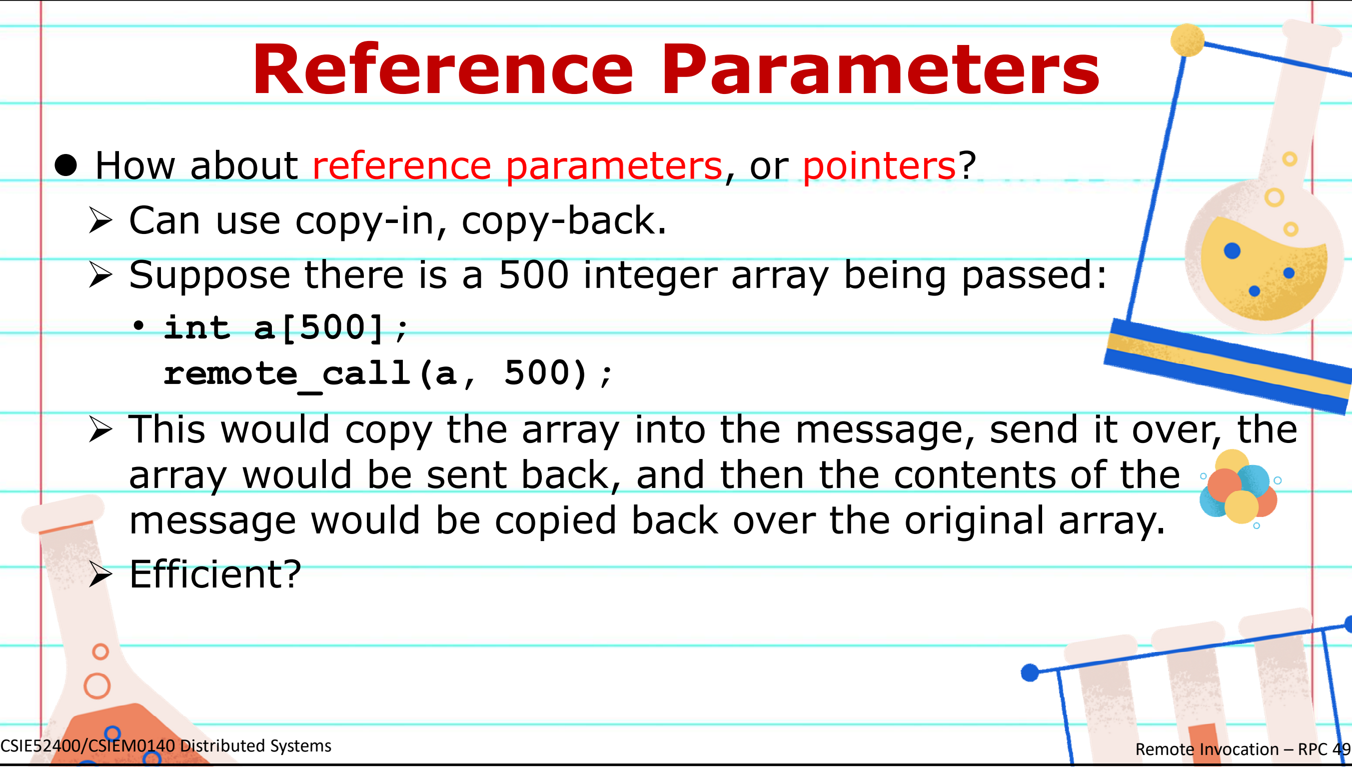
(a)

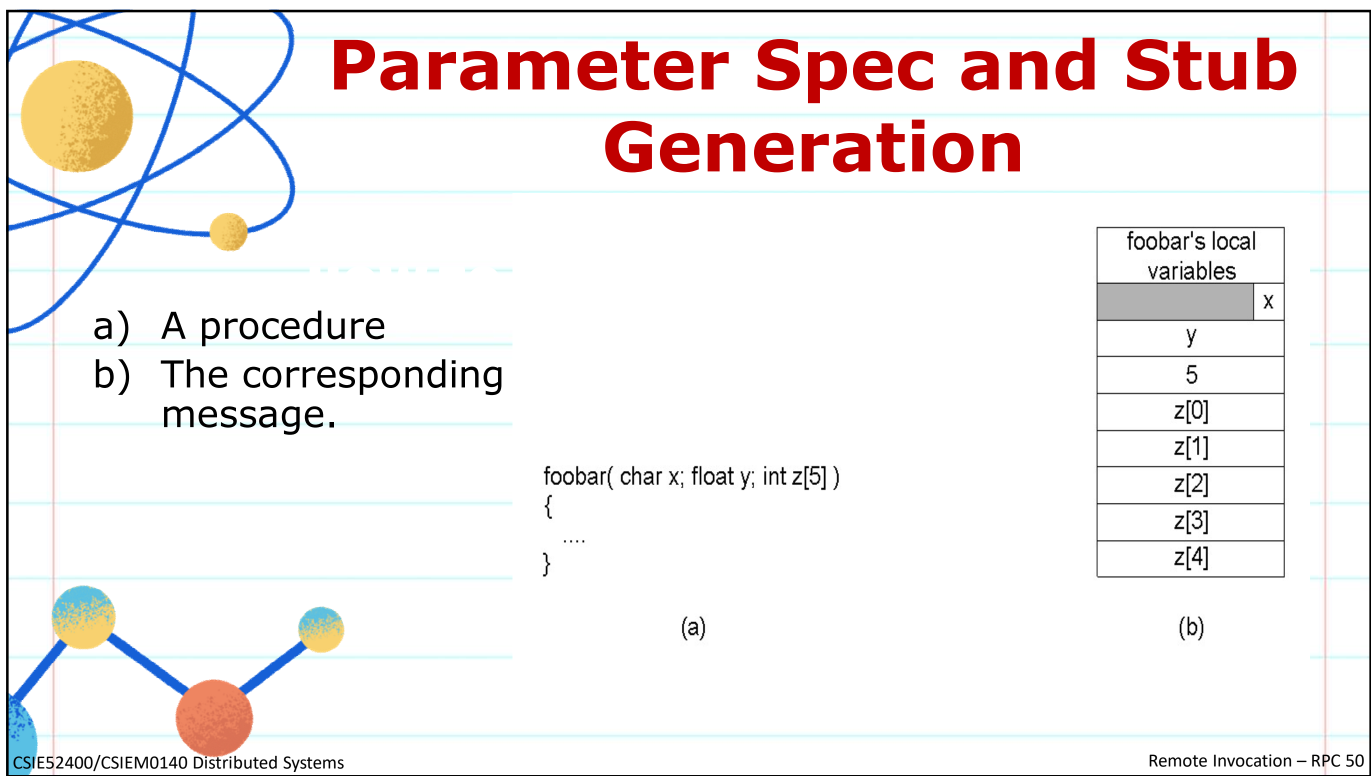
0	1	2	3
5	0	0	0
4	5	6	7
J	I	L	L

(b)

0	1	2	3
0	0	0	5
4	5	6	7
L	L	I	J

(c)


Reference Parameters
● How about reference parameters, or pointers?
➢ Can use copy-in, copy-back.
➢ Suppose there is a 500 integer array being passed:
• int a[500];
remote_call(a, 500);
➢ This would copy the array into the message, send it over, the array would be sent back, and then the contents of the message would be copied back over the original array.
➢ Efficient?
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 49


Parameter Spec and Stub Generation
a) A procedure
b) The corresponding message.
foobar(char x; float y; int z[5])
{
....
}
(a)
foobar's local variables
x
y
5
z[0]
z[1]
z[2]
z[3]
z[4]
(b)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 50

Where to bind?

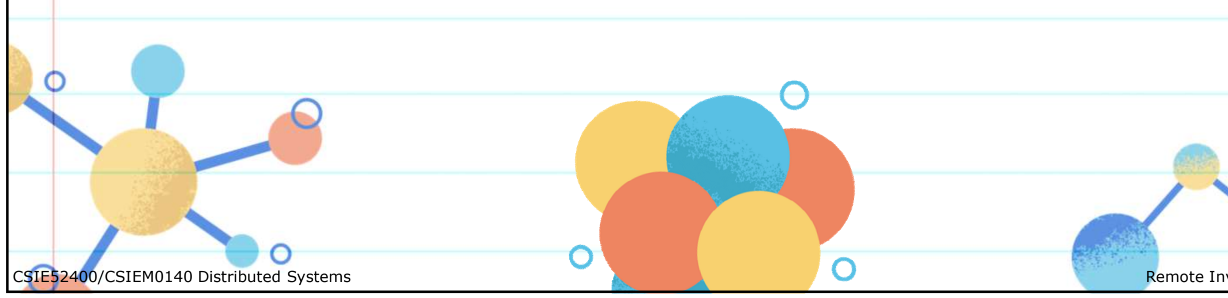
- Solution 1: Maintain a centralized DB that can locate a host that provides a particular service *(Birrell & Nelson's 1984 proposal)*

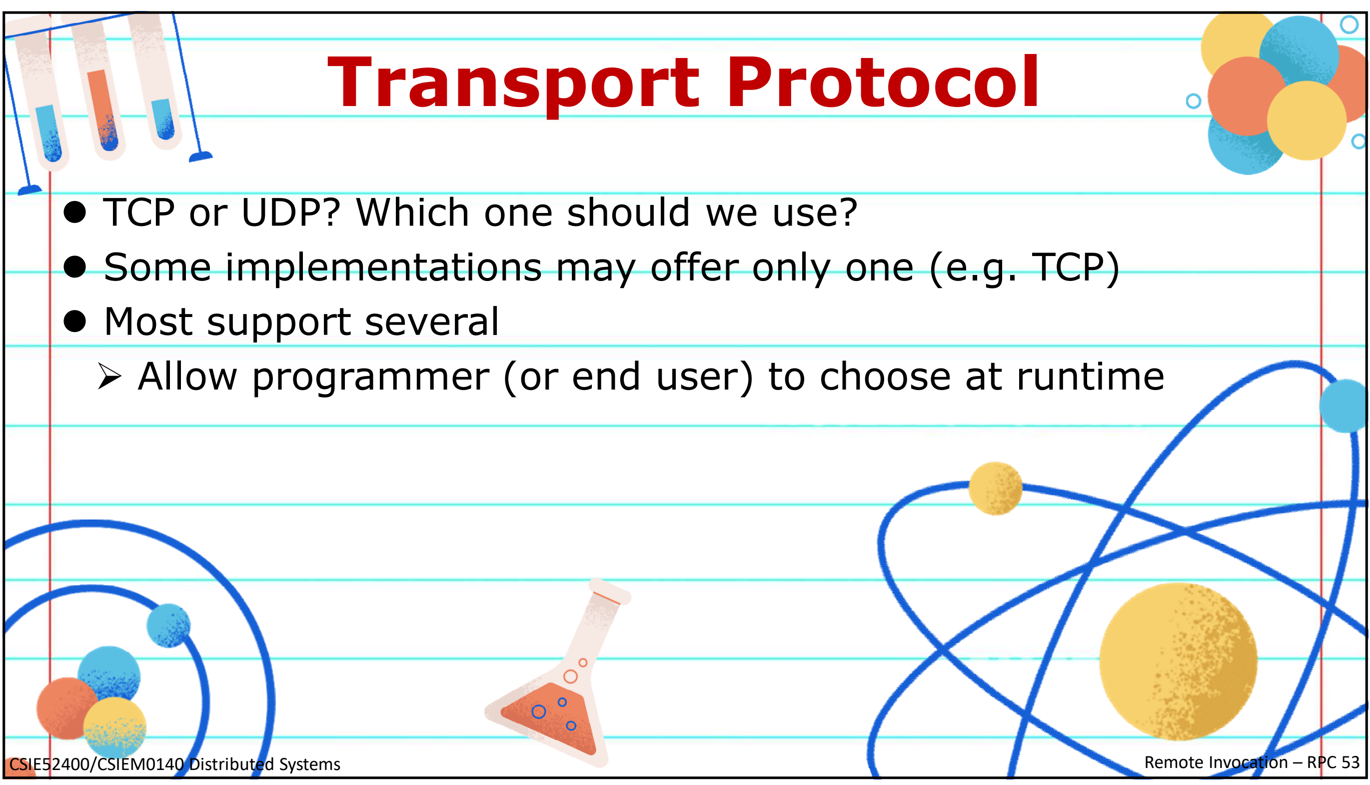
- Challenges:
  - Who administers this?
  - What is the scope of administration?
  - What if the same services run on different machines (e.g., file systems)?

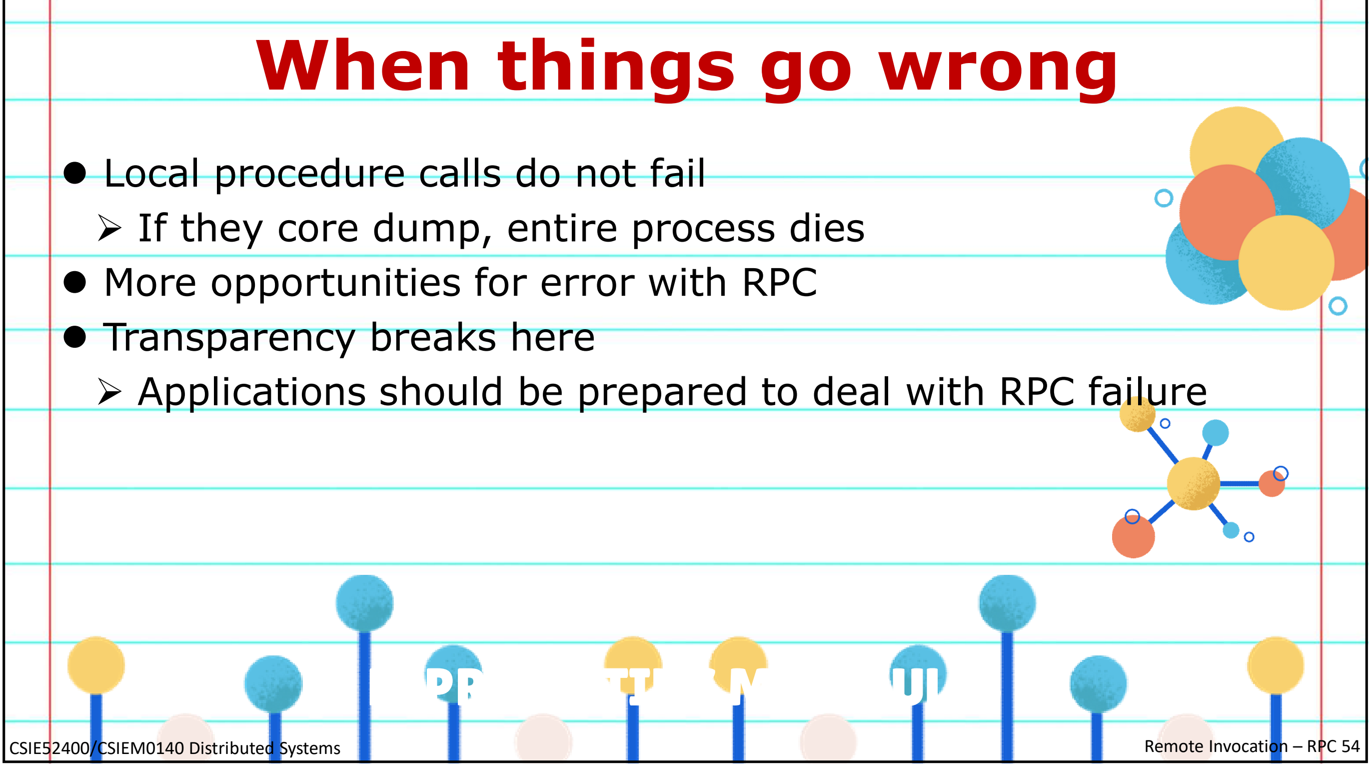
Where to bind?

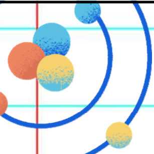
- Solution 2: A server on each host maintains a DB of *locally* provided services

- Challenges:
  - How to handle heterogeneous systems?
  - How to provide a uniform interface to the client?




Transport Protocol
● TCP or UDP? Which one should we use?
● Some implementations may offer only one (e.g. TCP)
● Most support several
➢ Allow programmer (or end user) to choose at runtime
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 53


When things go wrong
● Local procedure calls do not fail
➢ If they core dump, entire process dies
● More opportunities for error with RPC
● Transparency breaks here
➢ Applications should be prepared to deal with RPC failure
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 54



RPC Optimizations

- What if doing RPC to a process on the same machine?
  - Can use shared memory to optimize?
  - Can we get rid of all copies?
- Does RPC allow you to overlap communication with computation?
  - Suppose you are sending an array of 1 GB.
- Does RPC give you concurrency? That is, can you have the caller and callee executing at the same time?
  - Does normal procedure call provide that?

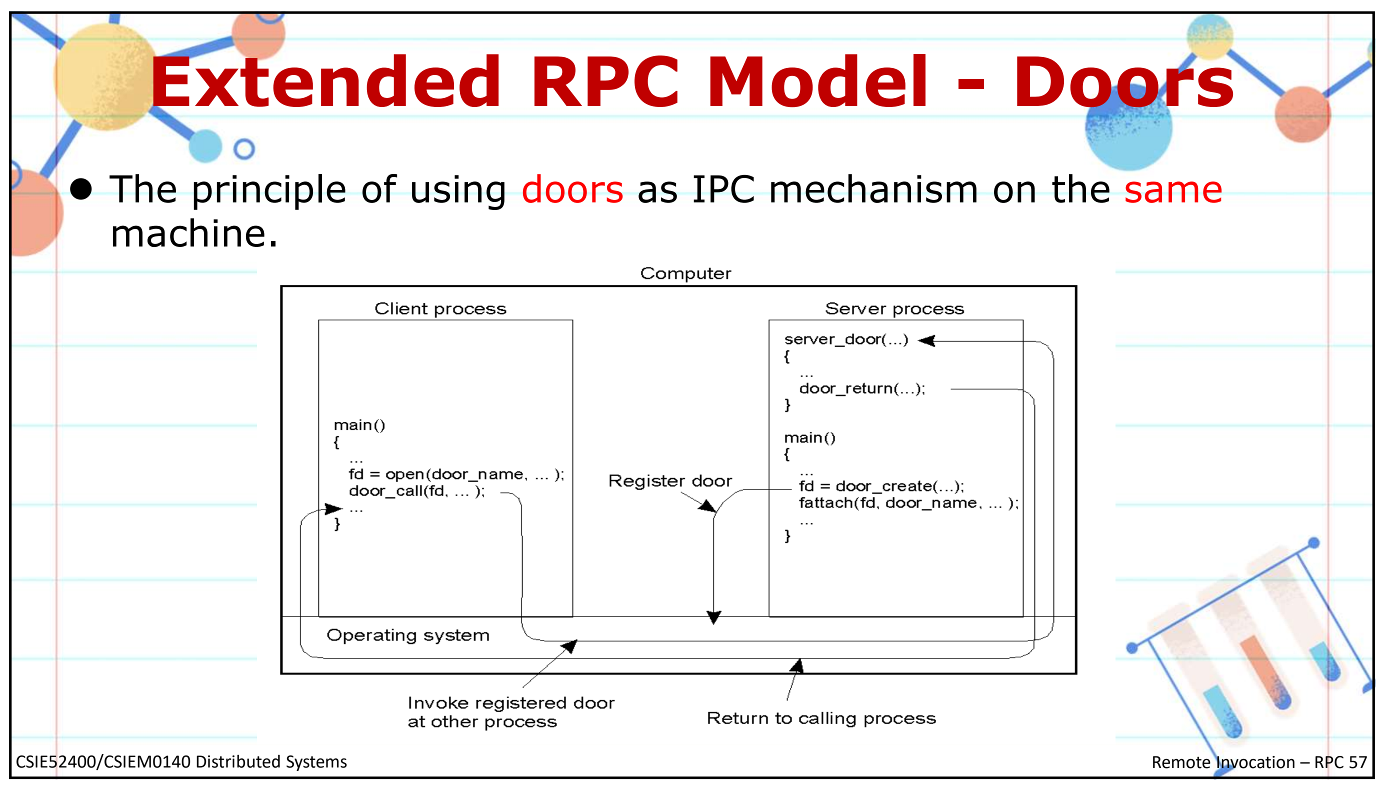
Using Shared Memory

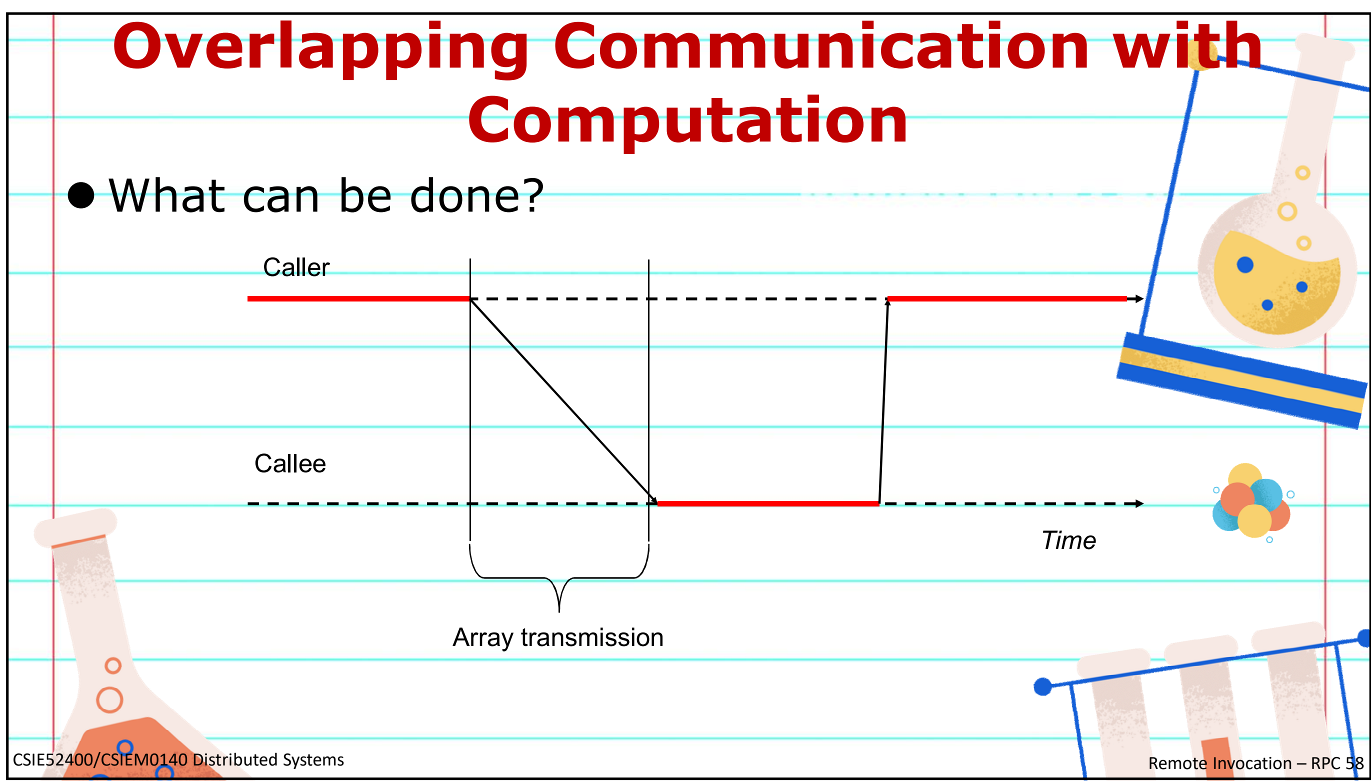
- Caller side:

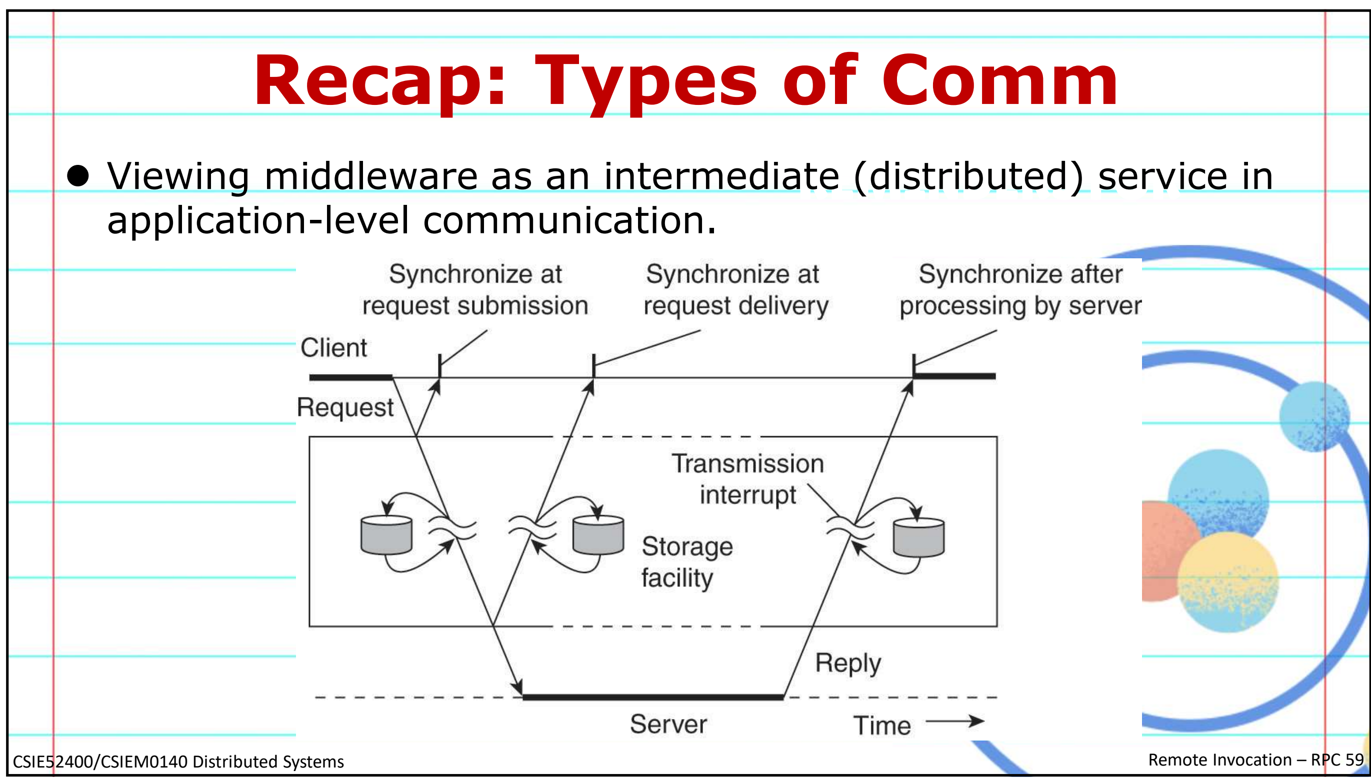
```
 a = func(big_array);
int func(int big_array[1000]) {
  copy_to_shared_memory(…);
  signal_callee(…);
  copy_result_from_shared_memory(…);
  return result;
}
```

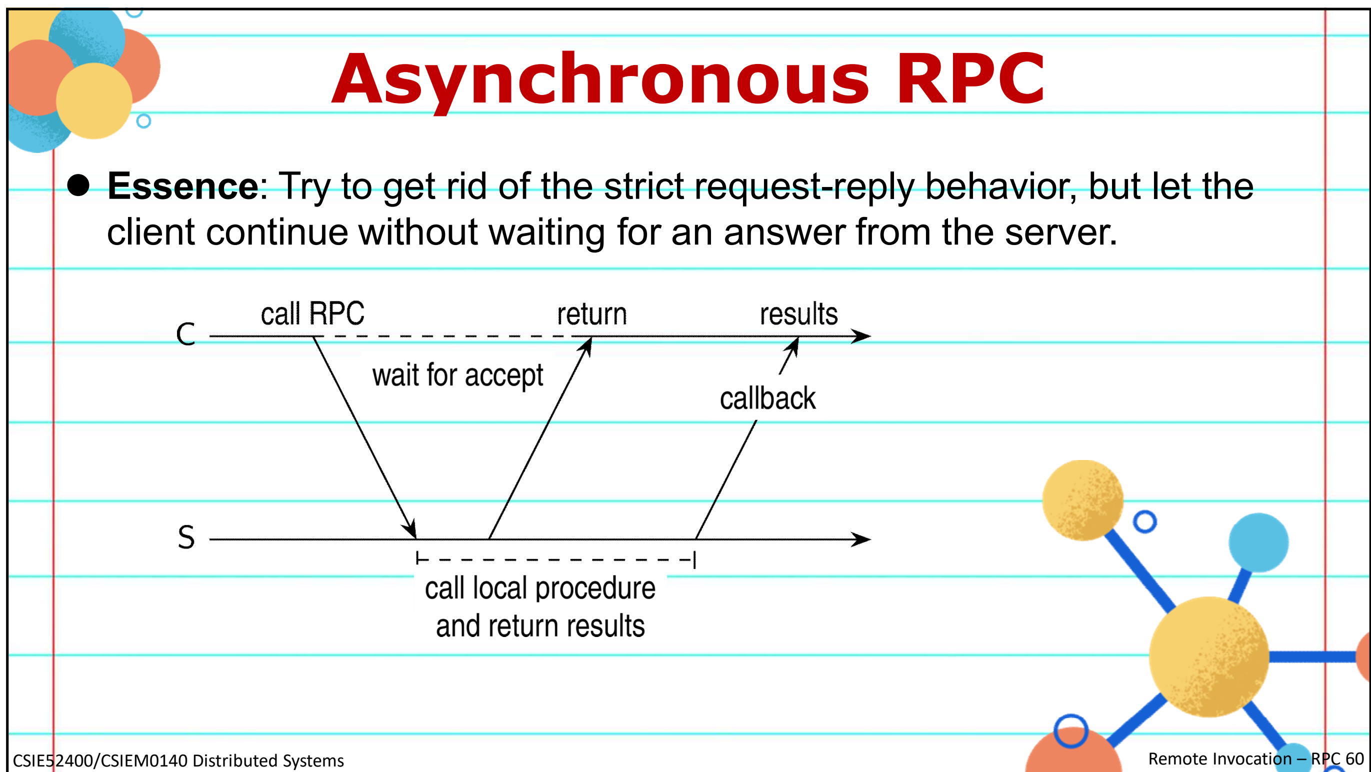
- Server side:

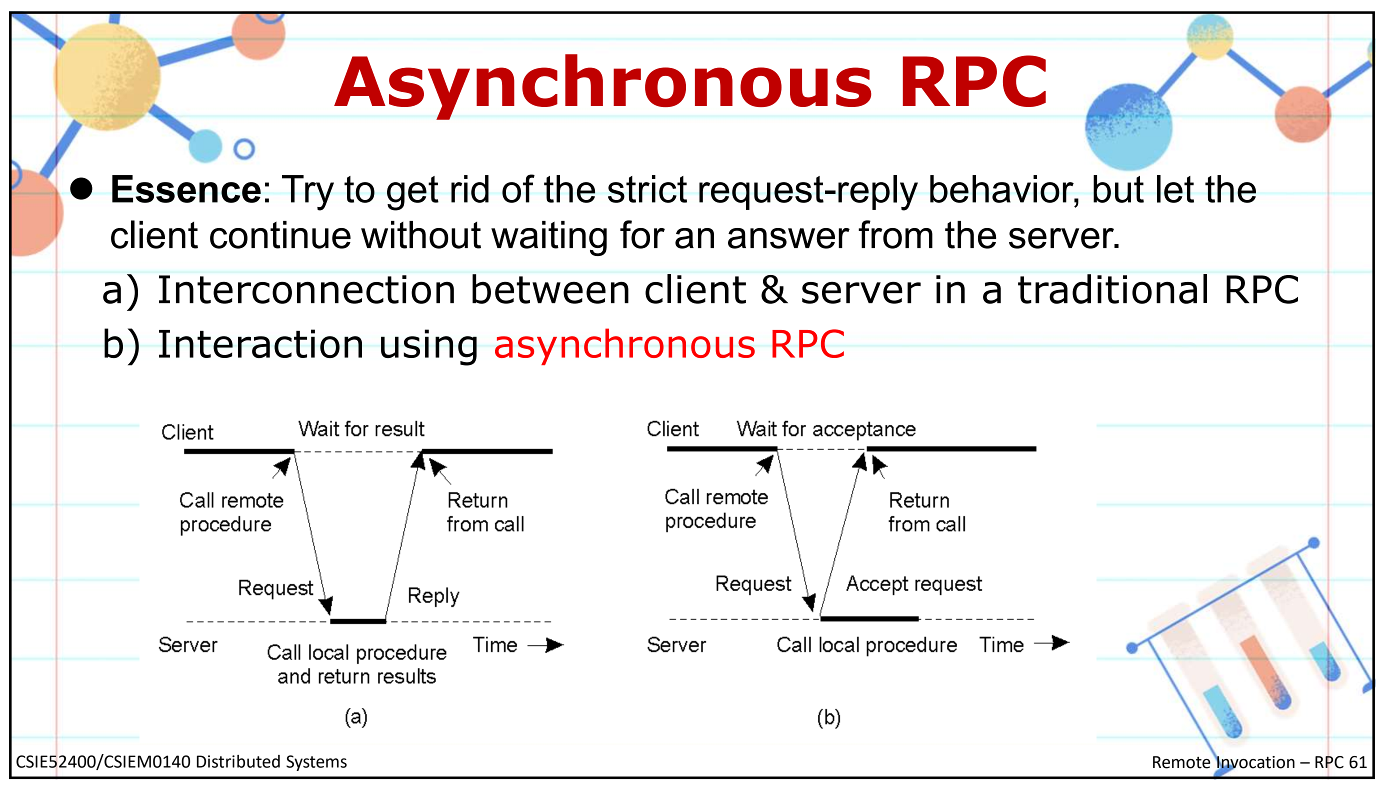
```
some_func(…) {
  wait_for_signal(…);
  call_local_func(…);
  /* Directly access from shared memory.*/
```

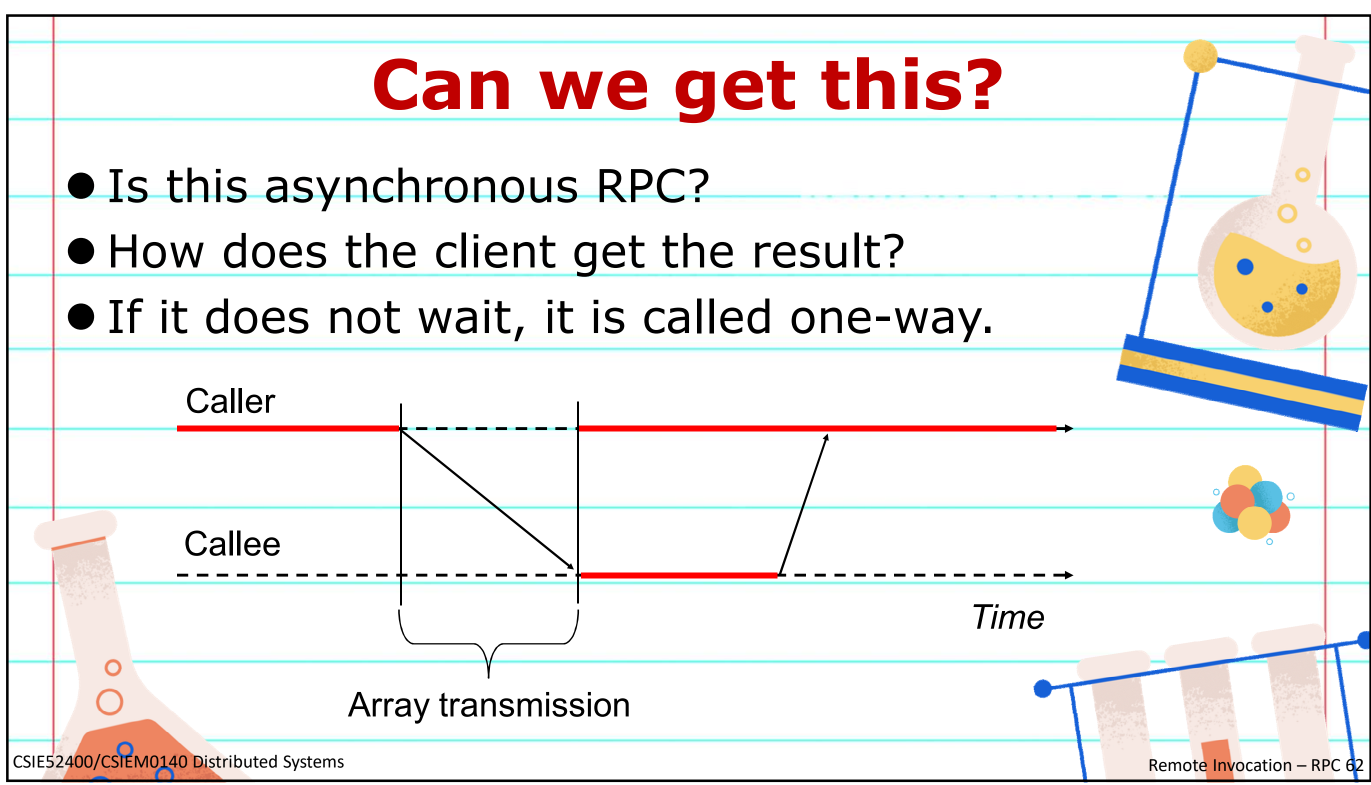

Extended RPC Model - Doors
The principle of using doors as IPC mechanism on the same machine.
Computer
Client process
Server process
main()
{
...
fd = open(door_name, ...);
door_call(fd, ...);
...
}
server_door(...)
{
...
door_return(...);
}
main()
{
...
fd = door_create(...);
fattach(fd, door_name, ...);
...
}
Register door
Operating system
Invoke registered door at other process
Return to calling process
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 57

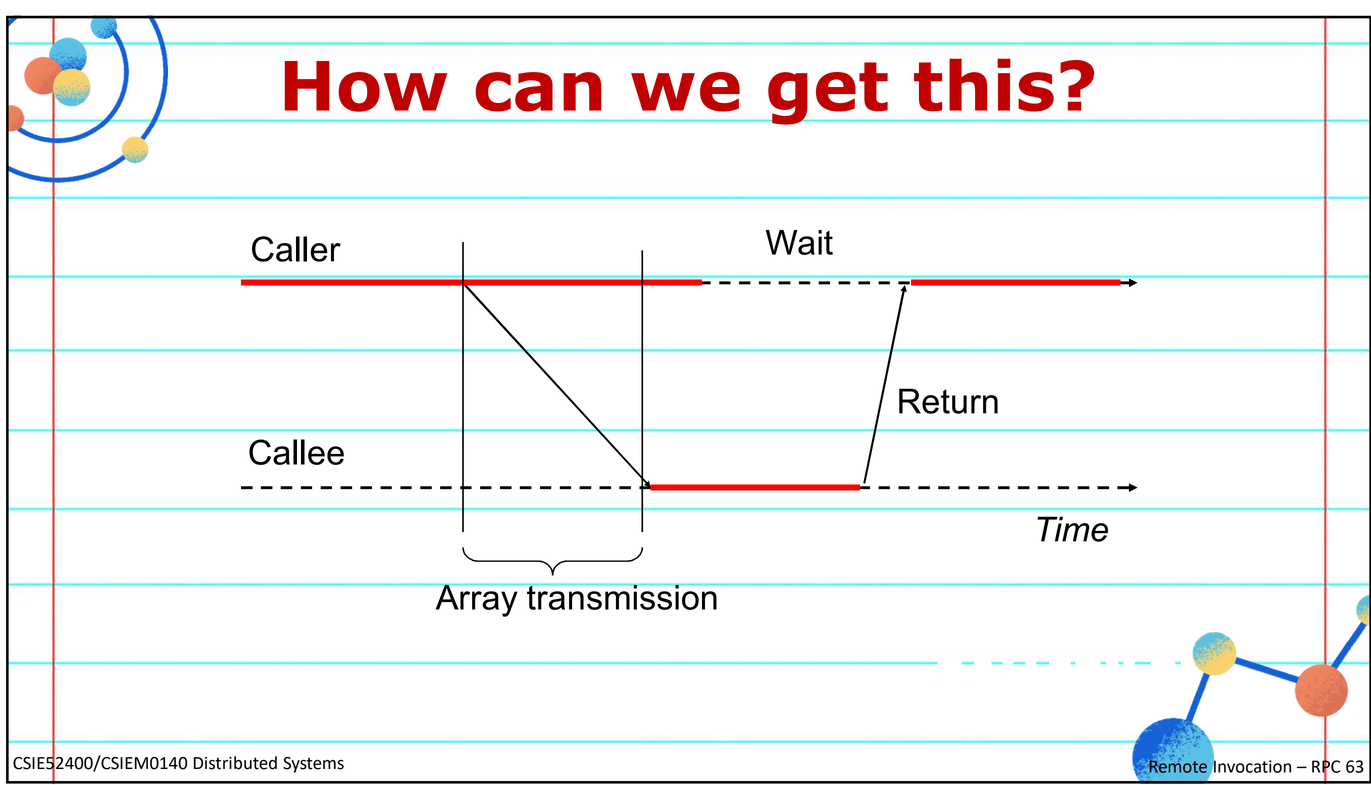

Overlapping Communication with Computation
What can be done?
Caller
Callee
Time
Array transmission
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 58

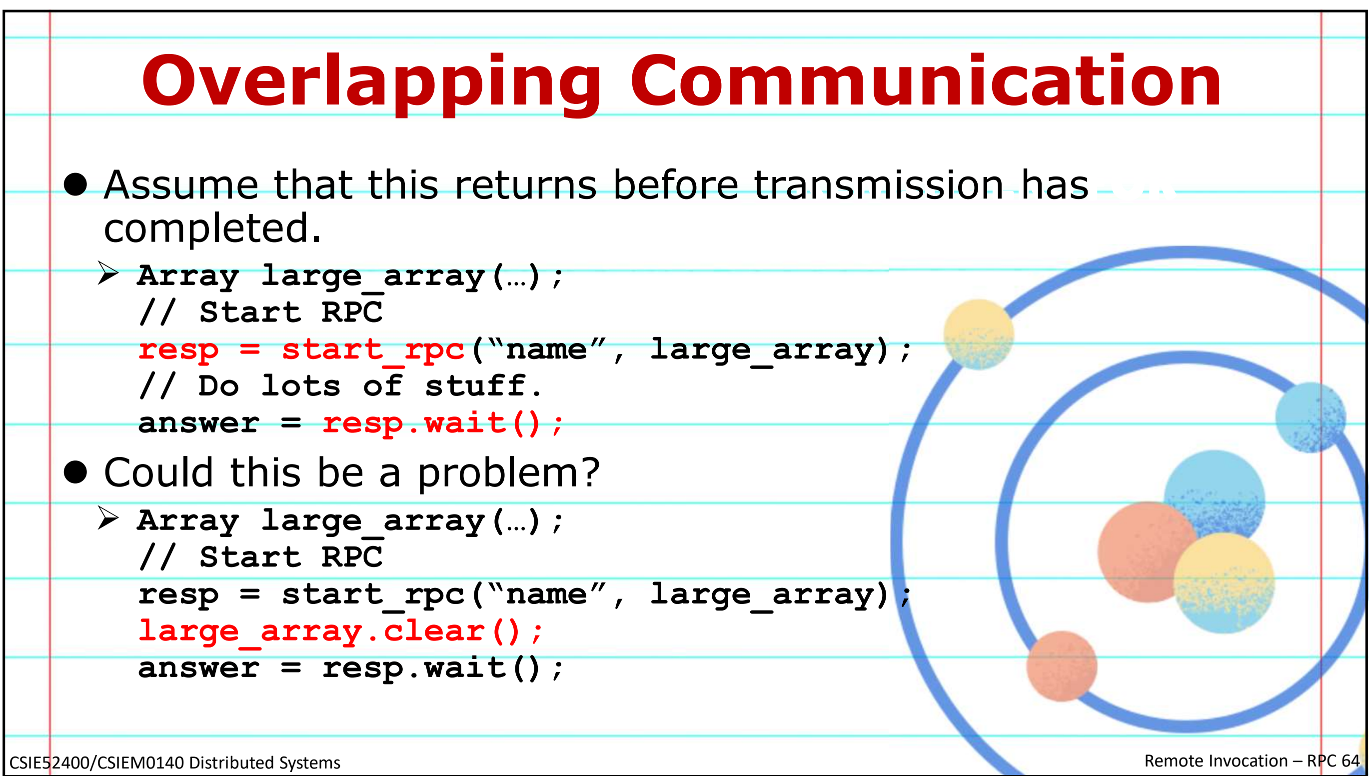

Recap: Types of Comm
Viewing middleware as an intermediate (distributed) service in application-level communication.
Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 59

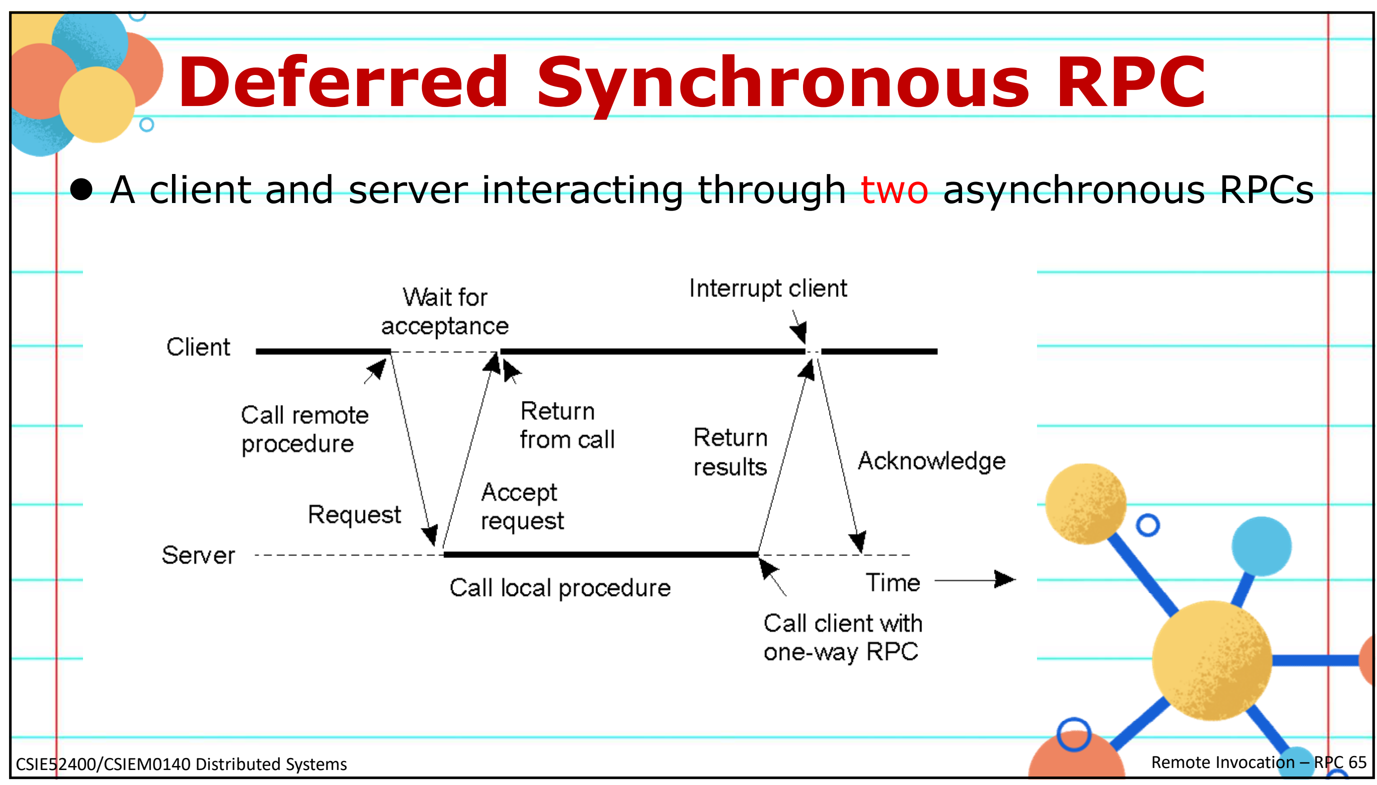

Asynchronous RPC
Essence: Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.
C
call RPC
return
results
wait for accept
callback
S
call local procedure and return results
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 60

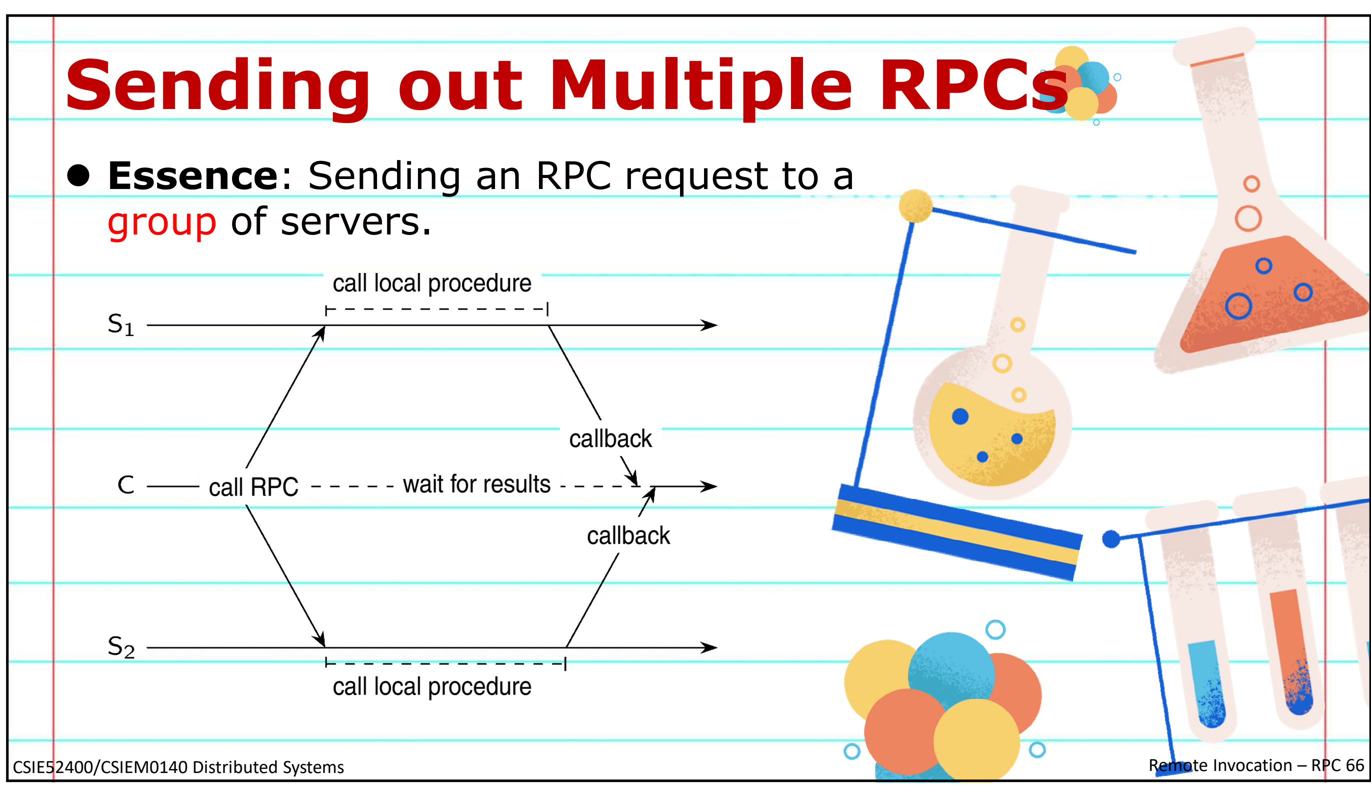

Asynchronous RPC
Essence: Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.
a) Interconnection between client & server in a traditional RPC
b) Interaction using asynchronous RPC
Client
Wait for result
Call remote procedure
Return from call
Request
Reply
Server
Call local procedure and return results
Time
(a)
Client
Wait for acceptance
Call remote procedure
Return from call
Request
Accept request
Server
Call local procedure
Time
(b)
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 61

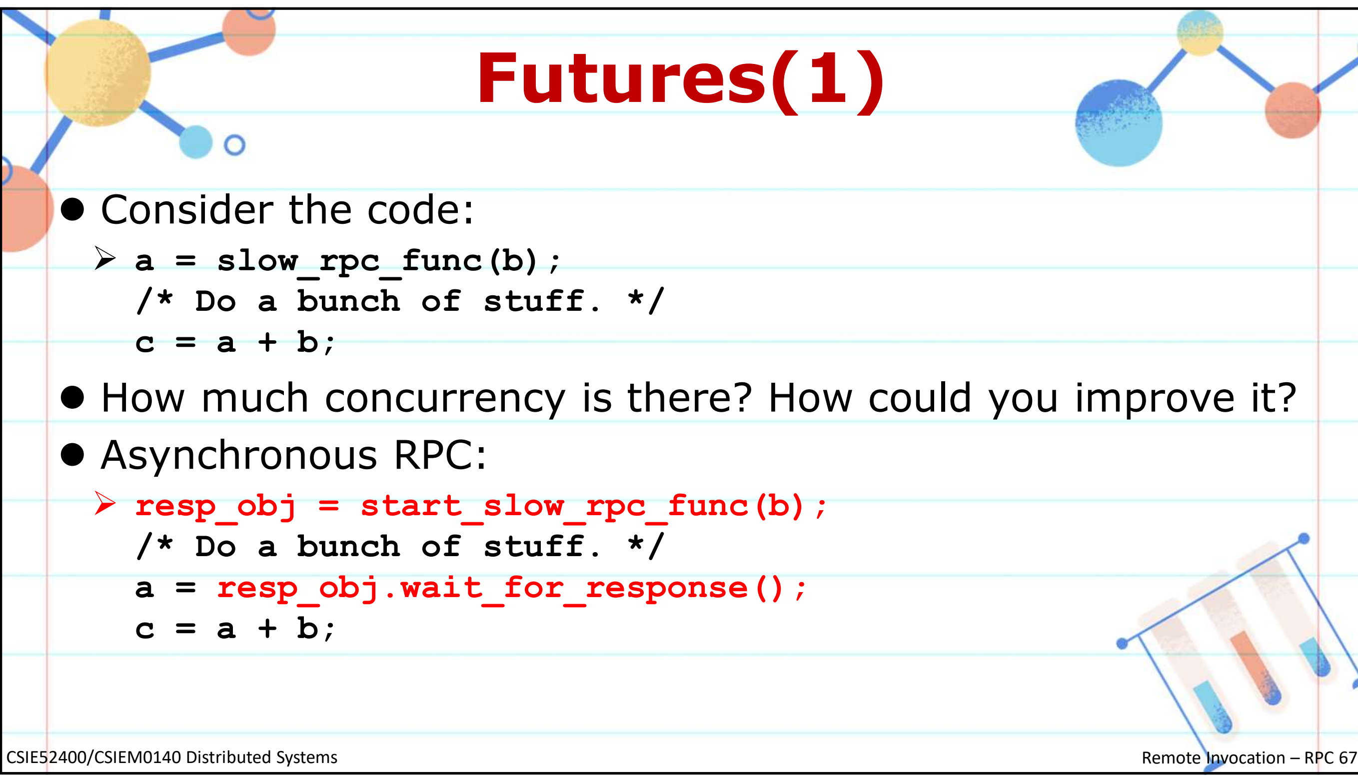

Can we get this?
Is this asynchronous RPC?
How does the client get the result?
If it does not wait, it is called one-way.
Caller
Callee
Time
Array transmission
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 62

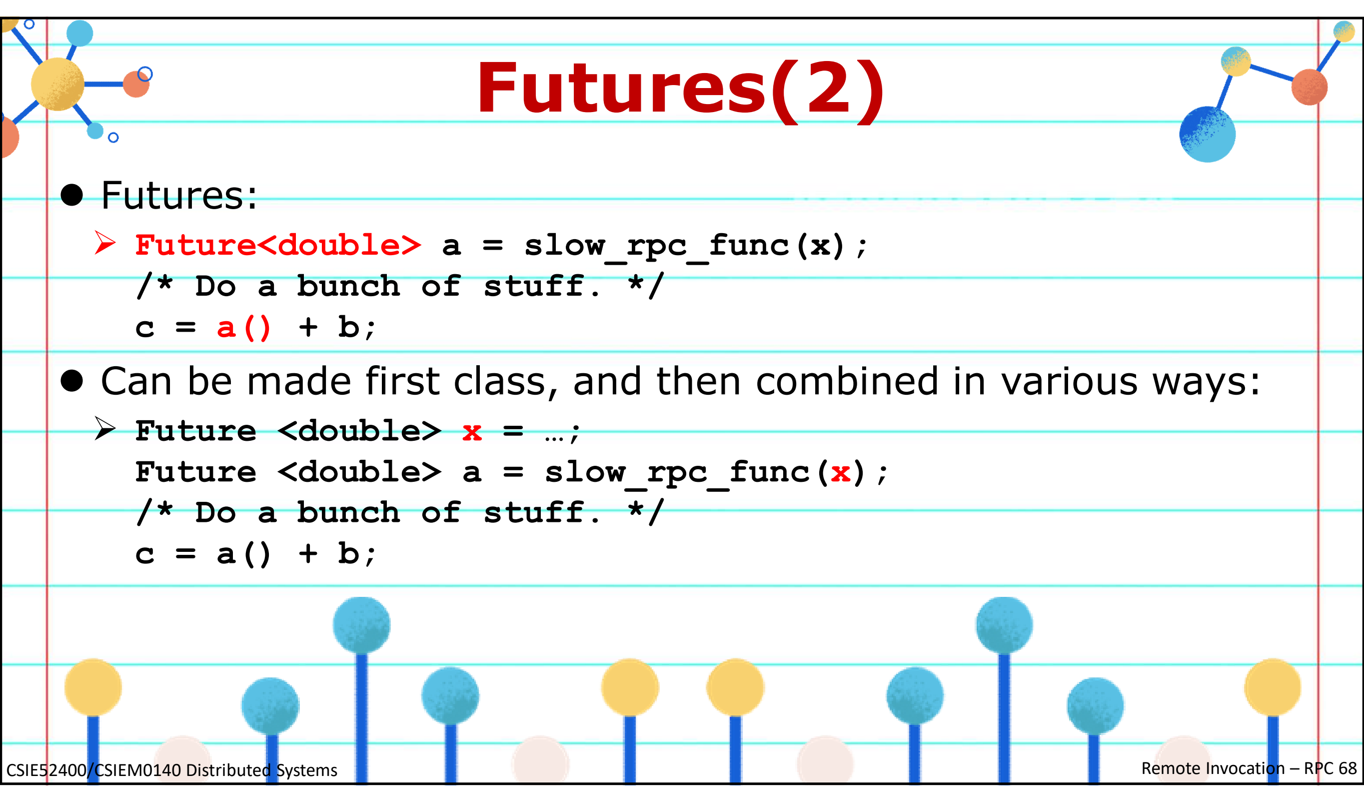

How can we get this?
Caller
Wait
Return
Callee
Time
Array transmission
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 63


Overlapping Communication
Assume that this returns before transmission has completed.
Array large_array(…);
// Start RPC
resp = start_rpc("name", large_array);
// Do lots of stuff.
answer = resp.wait();
Could this be a problem?
Array large_array(…);
// Start RPC
resp = start_rpc("name", large_array);
large_array.clear();
answer = resp.wait();
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 64


Deferred Synchronous RPC
A client and server interacting through two asynchronous RPCs
Wait for acceptance
Interrupt client
Client
Call remote procedure
Return from call
Return results
Acknowledge
Accept request
Request
Server
Call local procedure
Time
Call client with one-way RPC
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 65


Sending out Multiple RPCs
Essence: Sending an RPC request to a group of servers.
call local procedure
S1
callback
C
call RPC
wait for results
callback
S2
call local procedure
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 66


Futures(1)
● Consider the code:
➢ a = slow_rpc_func(b);
/* Do a bunch of stuff. */
c = a + b;
● How much concurrency is there? How could you improve it?
● Asynchronous RPC:
➢ resp_obj = start_slow_rpc_func(b);
/* Do a bunch of stuff. */
a = resp_obj.wait_for_response();
c = a + b;
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 67

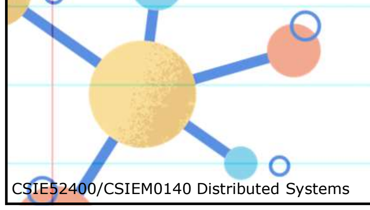

Futures(2)
● Futures:
➢ Future<double> a = slow_rpc_func(x);
/* Do a bunch of stuff. */
c = a() + b;
● Can be made first class, and then combined in various ways:
➢ Future <double> x = …;
Future <double> a = slow_rpc_func(x);
/* Do a bunch of stuff. */
c = a() + b;
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 68

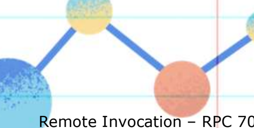
RPC Example – DCE/RPC

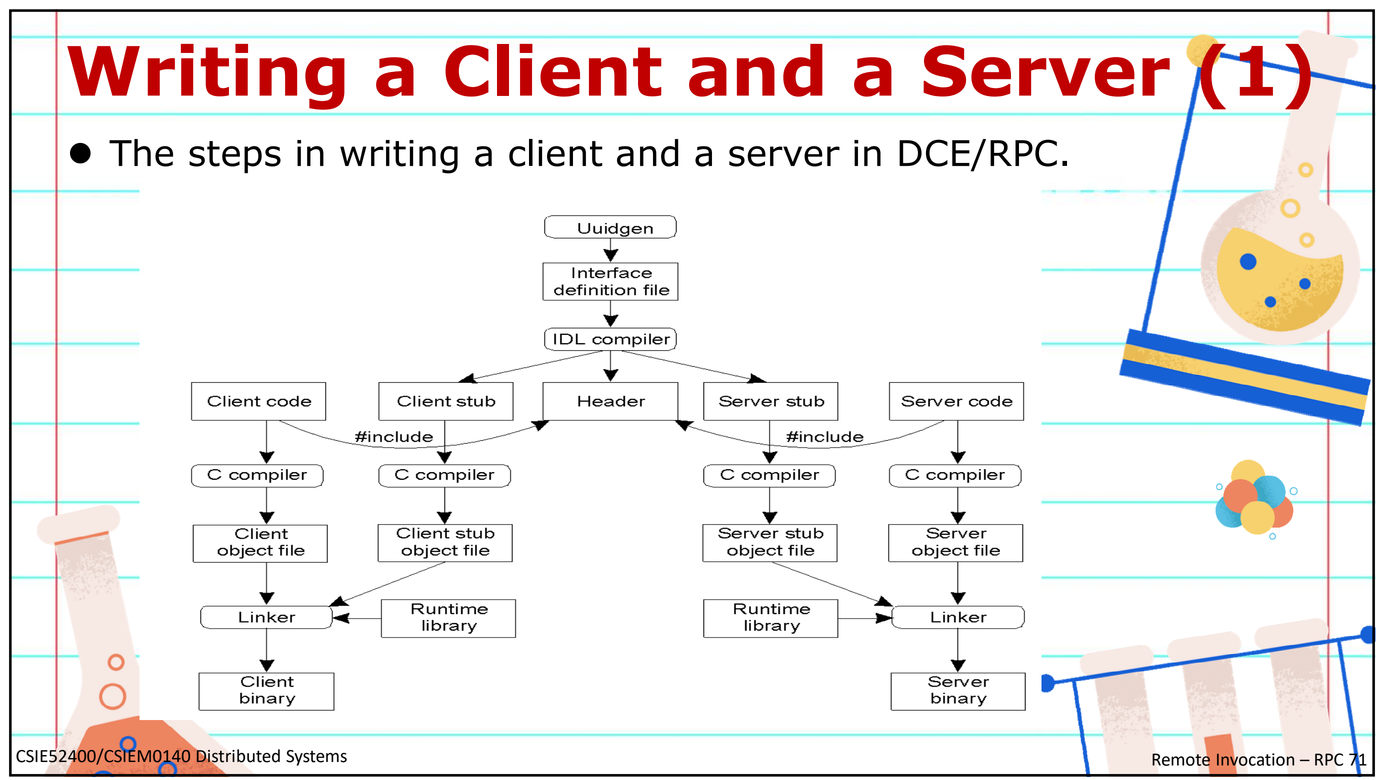
- DCE (Distributed Computing Environment) is developed by **Open Software Foundation** (**OSF**, now called **The Open Group**) as an open middleware for distributed systems.
- The key idea is to add an open distributed layer easily w/o disturbing existing applications.
- Based on client-server model
- Representative of typical RPC systems.
- Provide a number of services:
  - Distributed file service (transparent file access)
  - Directory service (transparent resource access)
  - Security service (resource protection)
  - Distributed time service (keep clocks globally synchronized)

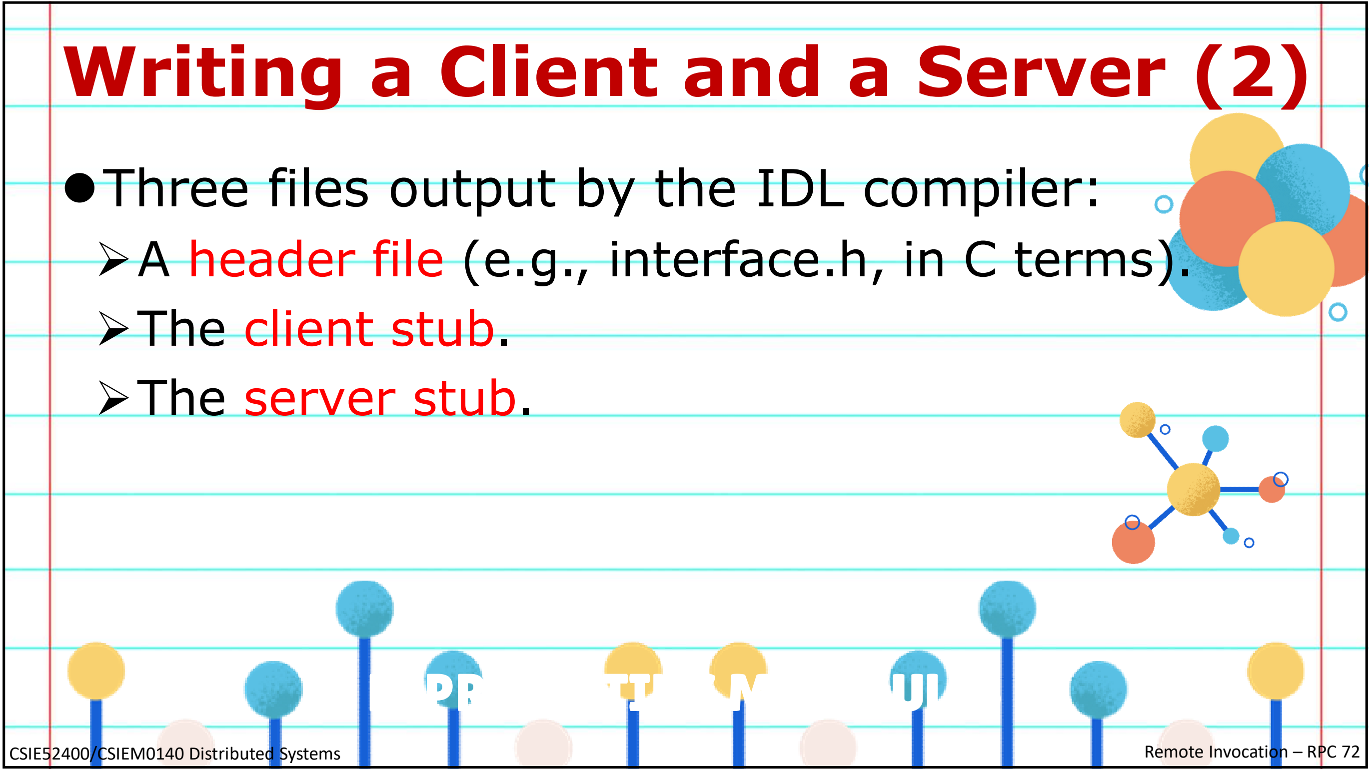
Goals of DCE/RPC

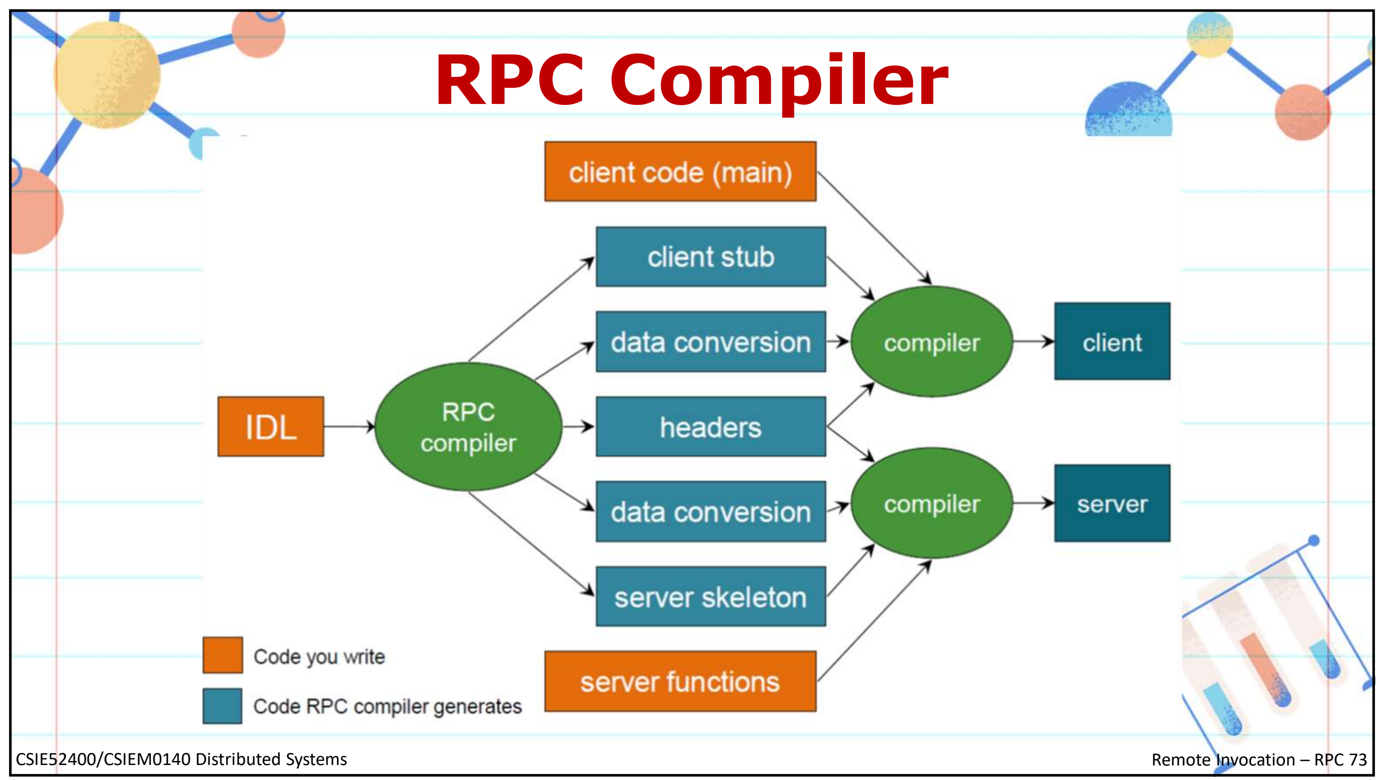
- A client can access a remote service by calling a local procedure.
- Existing code run in DCE with few or no changes.
- Automatically locate the correct server and set up the communication (binding).
- Handle two-way message communication.
- Handle data type conversions
- Clients and servers can be written using different languages, run on different platforms.

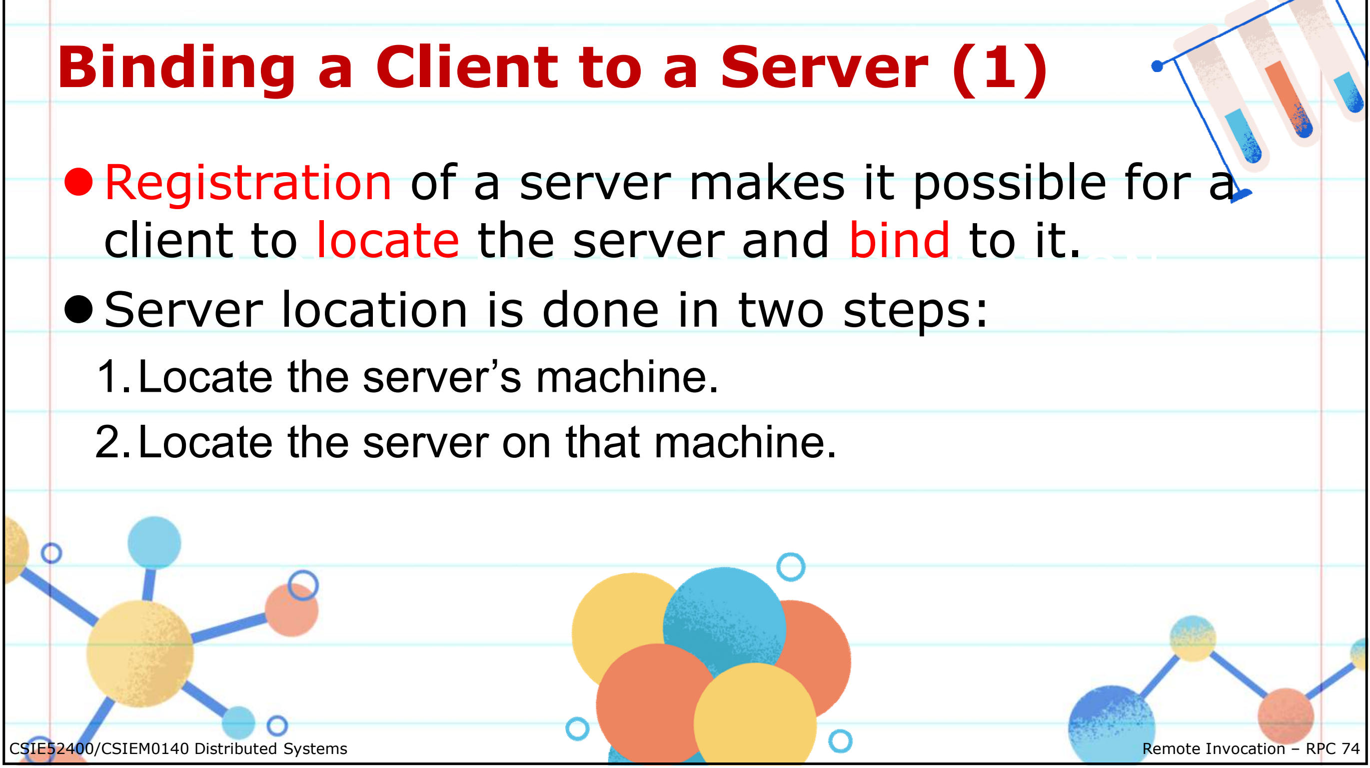


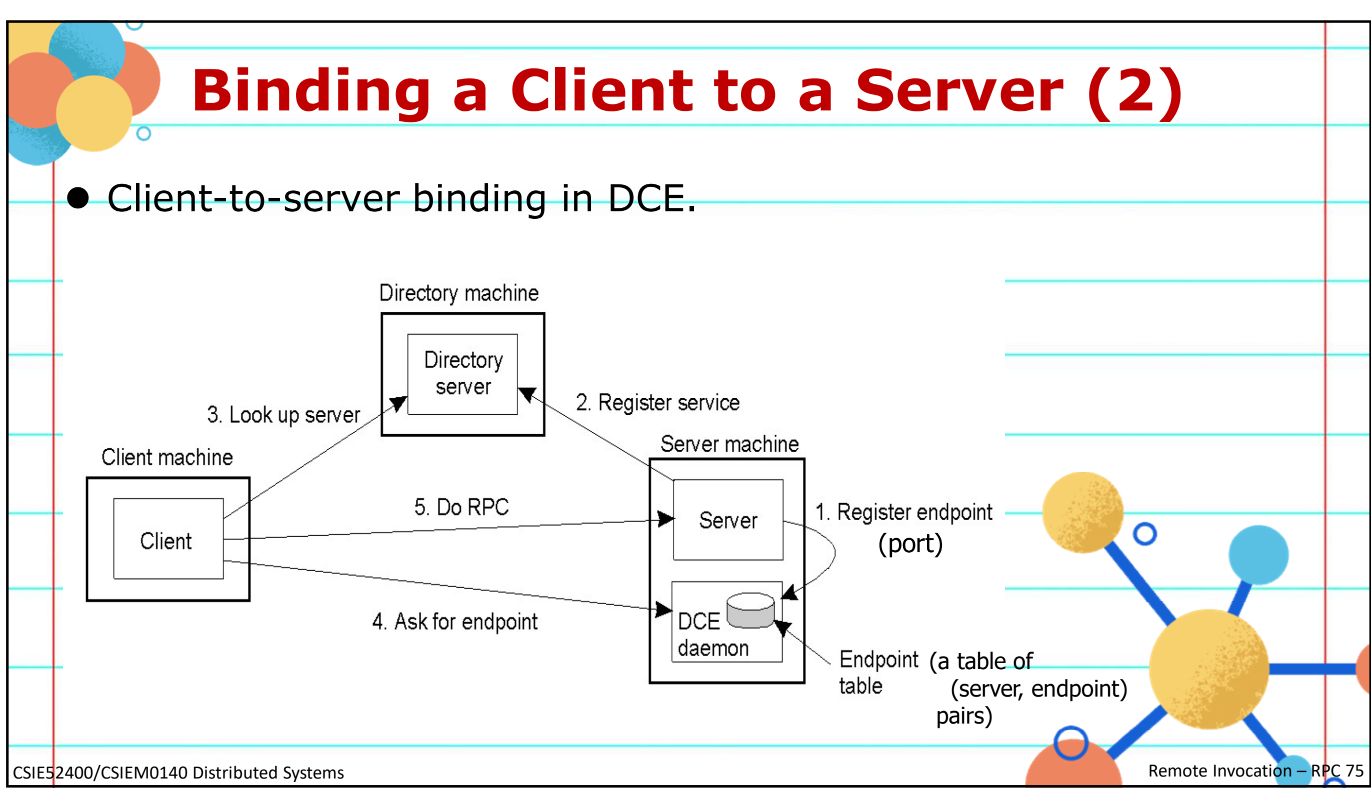



Writing a Client and a Server (1)
● The steps in writing a client and a server in DCE/RPC.
Uuidgen
Interface definition file
IDL compiler
Client code
Client stub
Header
Server stub
Server code
#include
#include
C compiler
C compiler
C compiler
C compiler
Client object file
Client stub object file
Server stub object file
Server object file
Linker
Runtime library
Runtime library
Linker
Client binary
Server binary
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 71


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.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 72

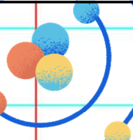

RPC Compiler
client code (main)
client stub
data conversion
headers
data conversion
server skeleton
server functions
IDL
RPC compiler
compiler
compiler
client
server
Code you write
Code RPC compiler generates
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 73


Binding a Client to a Server (1)
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.
CSIE52400/CSIEM0140 Distributed Systems
Remote Invocation – RPC 74



Other RPC Related Sys/Protocols

- FreeDCE
- MSRPC (Microsoft)
- J-Interop (MSRPC implementation in Java)
- Jarapac (DEC/RPC in Java)
- MS-RPCE (Microsoft Remote Procedure Call Protocol Extensions)
- JSON-RPC (a "JSON encoded" RPC variant)
- XML-RPC (an "XML encoded" RPC variant)
- SOAP is a successor of XML-RPC and also uses XML to encode its HTTP-based calls.



Python RPC Libraries

- Many libraries exist for Python RPC
- RPyC (are-pie-see), or Remote Python Call, is a transparent python library for symmetrical remote procedure calls.
- gRPC is an open source RPC system initially developed at Google.
- The Google Protocol RPC library is a framework for implementing HTTP-based RPC services.
- Apache Thrift (developed at Facebook) is a RPC framework for defining and creating services for numerous languages.
- See also: xmlrpc, json-rpc, tinyrpc