Remote invocation, Indirect communication & Web services.

CDK 5.1–5.2, 6.1, 6.3-6.4, 7.4, (9.1-9.4)

MDS E2015
Søren Debois

Meta

Misc

- You are not expected to work on MP2 in the fall break.
- More snapshot? (Frederik & Holger says you’re doing fine.)
- You should tell me to slow down.
- No exercises friday.

Summary

Security

- Goals
  (Integrity, confidentiality, availability, accountability)

- Assumptions
  (Attacker has control of network; can’t break crypto)

- Defenses
  (Crypto. Hard, though: for all/exist, social engineering.)

Cryptography

- Hashes
- Symmetric encryption schemes
- Asymmetric encryption schemes
- Signatures
- Certificates
- SSL/TLS

In the wild

- Man-in-the-middle attacks.
- Government bulk data collection.
- Ramifications of social media.

Plan

- Threads & synchronisation
- Indirect communication
- Mini-project 2

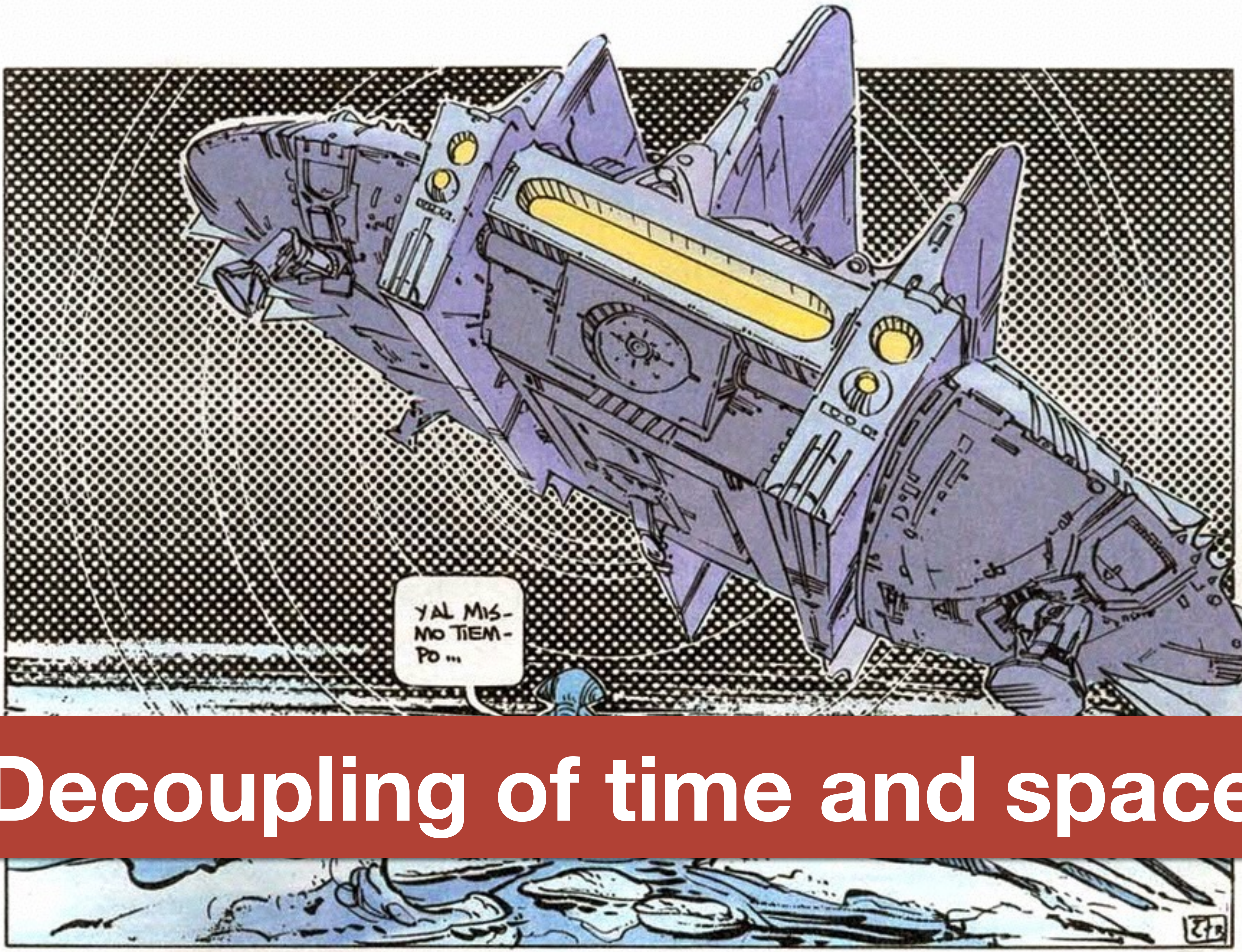
Threads & Synchronization

```
myList.add(x)

// Not concurrent with:

for (Object x : myList) {
  ...
}
```

NB! Datastructures

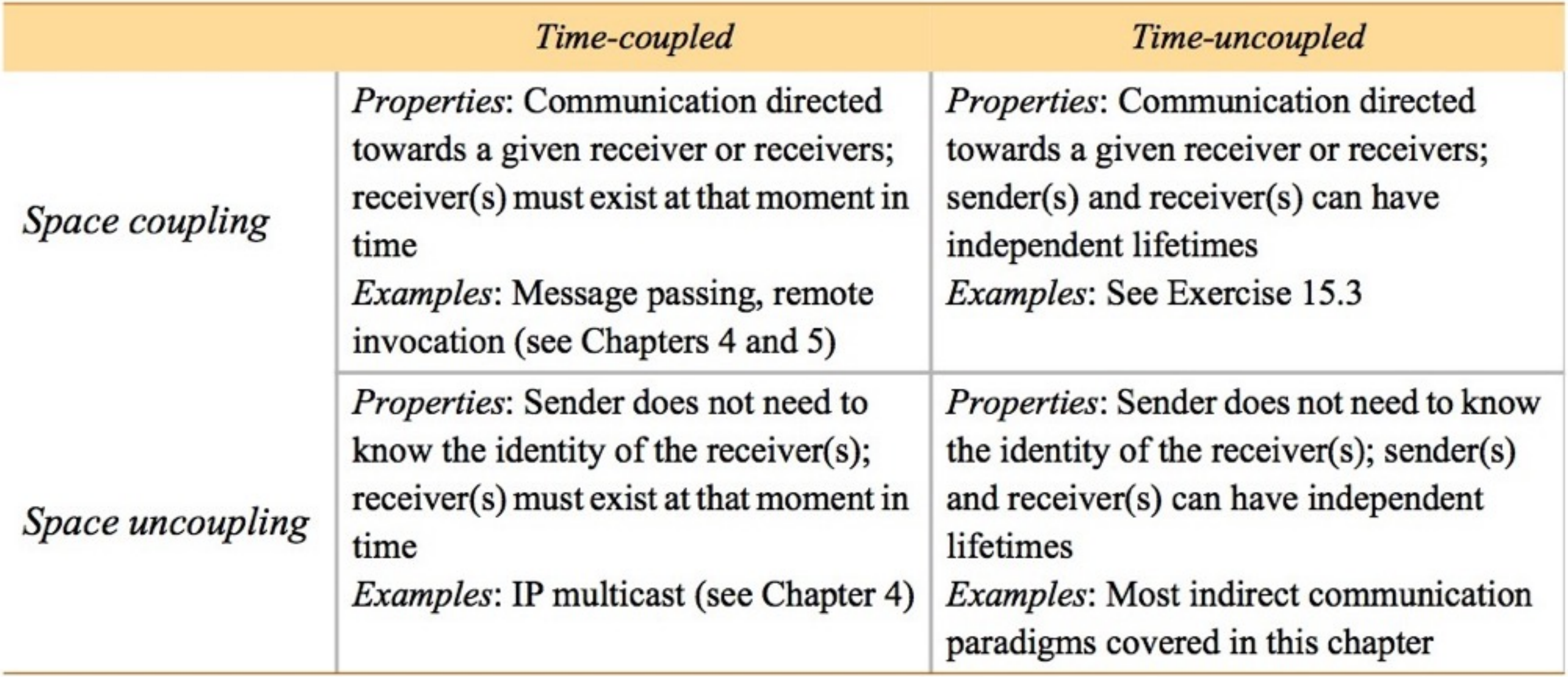
They read and write internally.

Indirect communication



Decoupling of time and space

Figure 6.1
Space and time coupling in distributed systems



	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 15.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

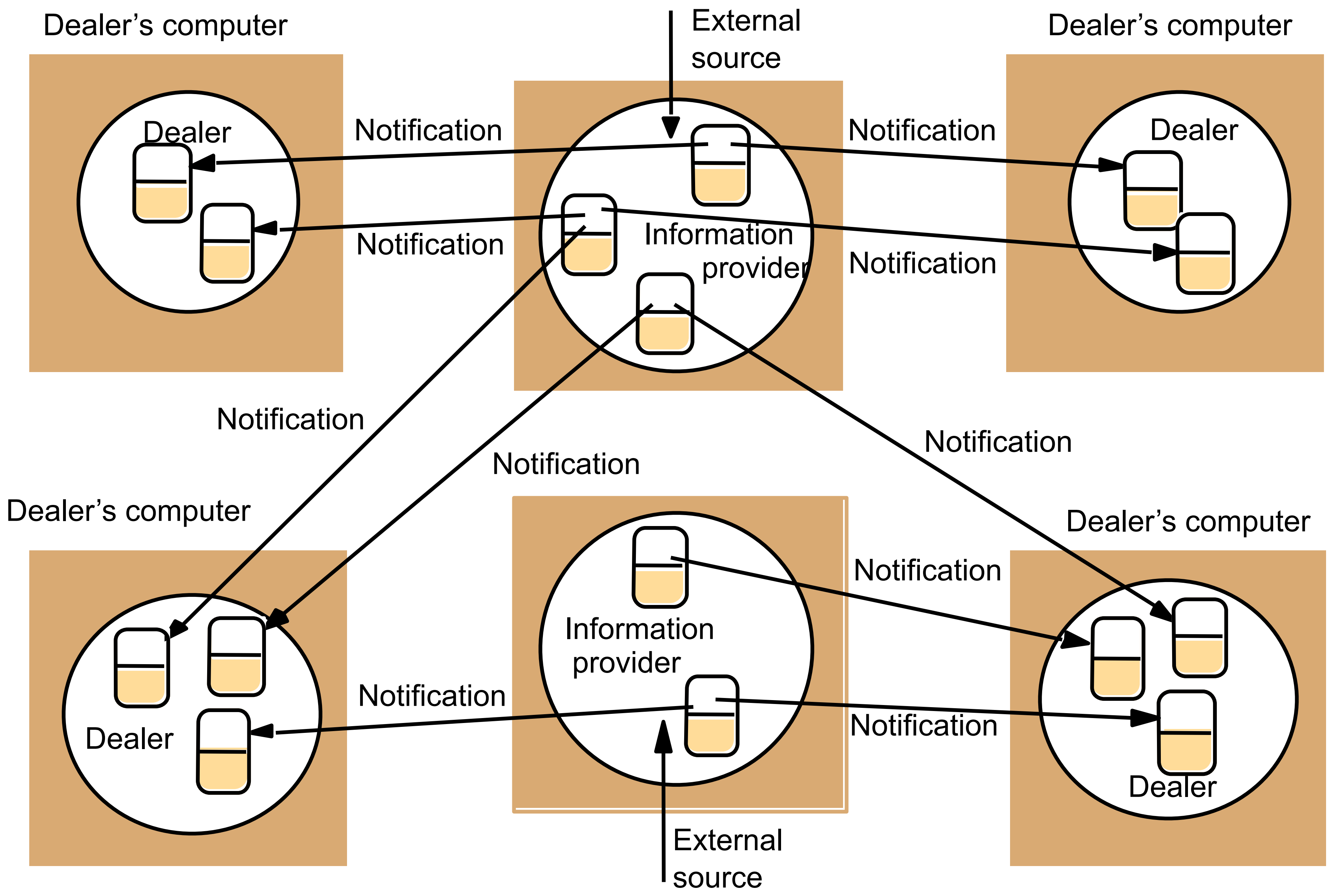
Space de-coupling

Don’t know the other guys’ identity

Time de-coupling

Sender and receiver don’t need to exist at the same time

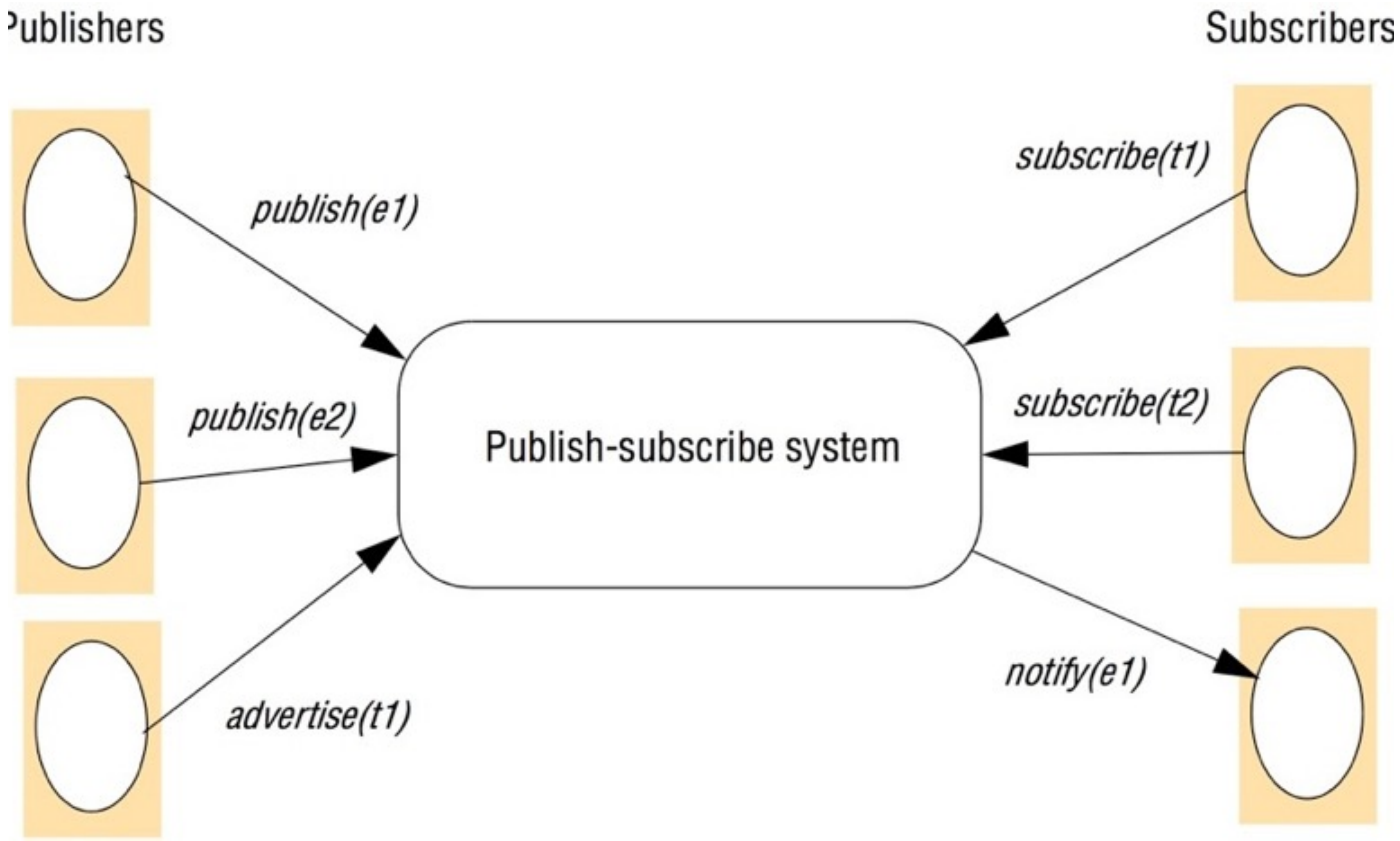
Publish/subscribe



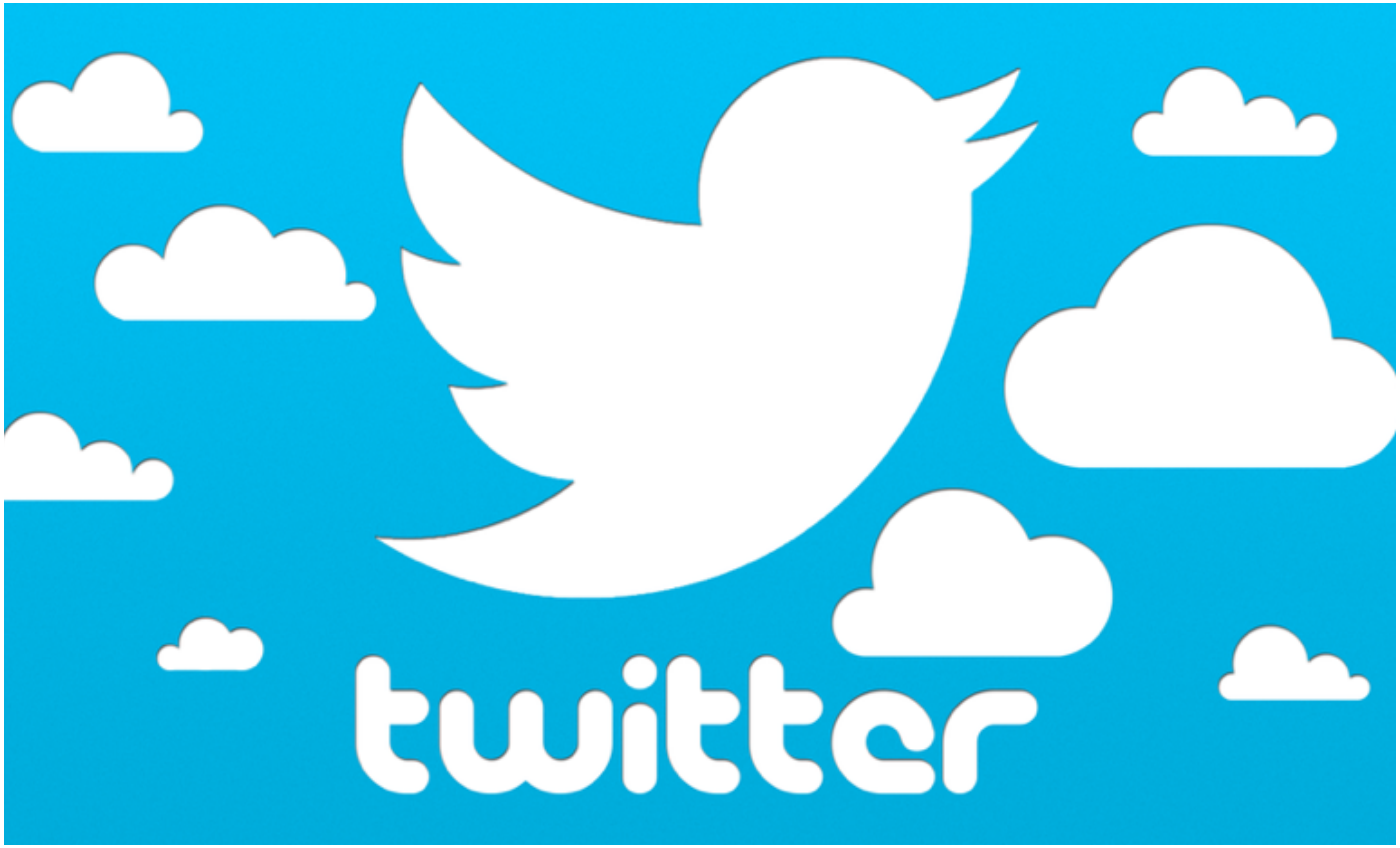
Publish/subscribe

Pub/sub programming model

- *Publisher*: publish(e)
- *Subscriber*: subscribe(f) / unsubscribe(f)
- *Subscriber*: notify(e)
- *Publisher:* advertise(f) / unadvertise(f)



Pub/sub programming model

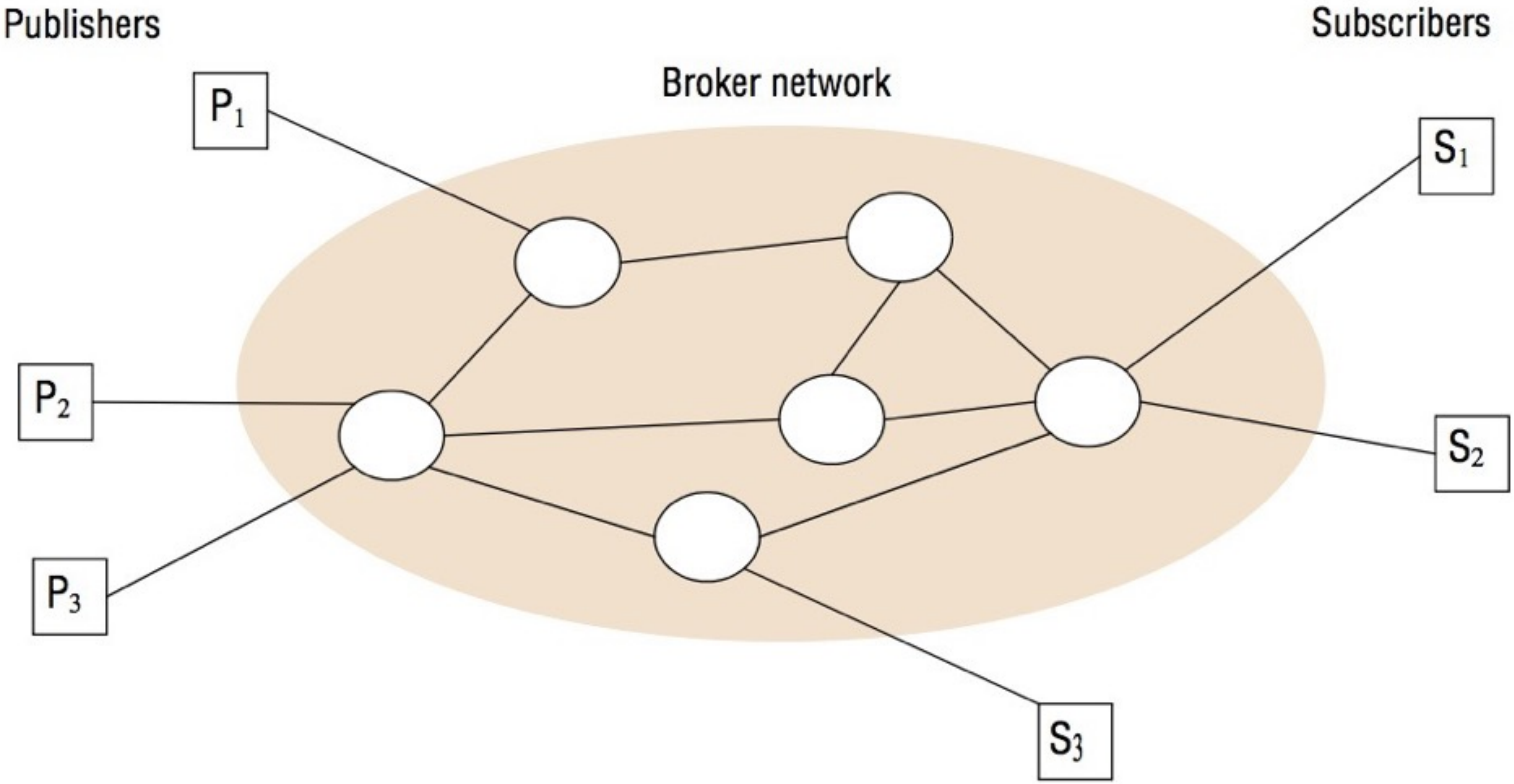


Example

Implementations

- **Centralised—client/server. (Duh.)**
- Multicast
- Overlay networks

Figure 6.9
A network of brokers



Summary

- Time-coupled
  Sender and receiver must exist at the same time.

- Space-coupled
  Sender and receiver must know each other

- Publish-subscribe
  API, Central server–multicast–overlay, Twitter

Read on your own

- Lots and lots of detail, especially implementation
- Message queues

Remote invocation

Request-reply

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

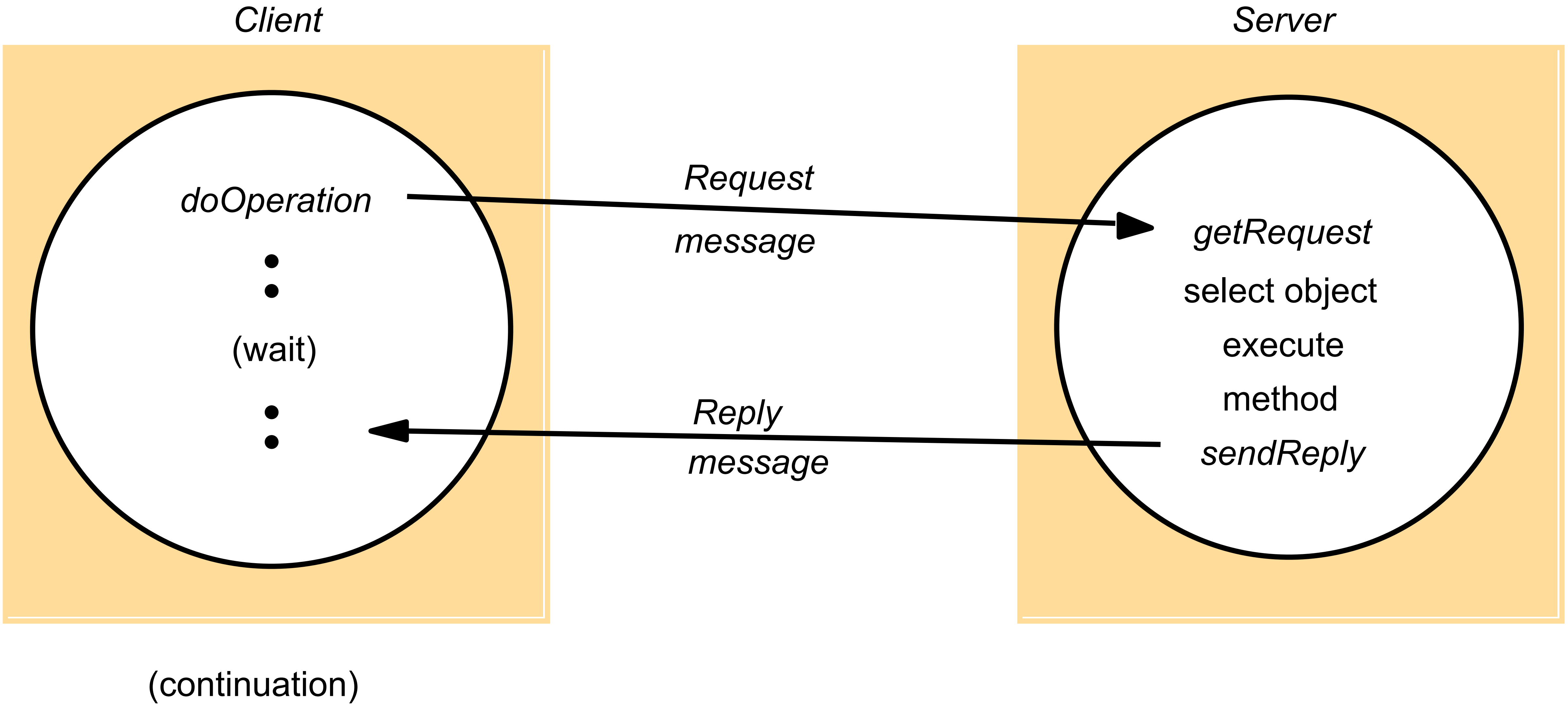
sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();

acquires a client request via the server port.

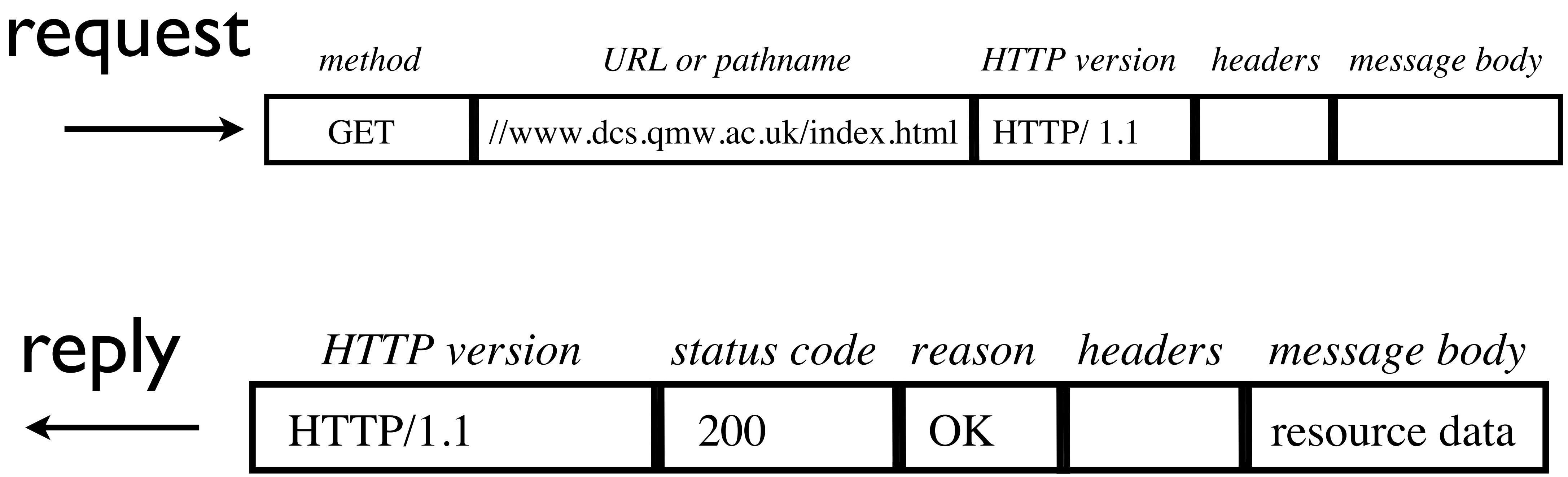
public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

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



Why is RR not trivial?

Omission failures (req & rep), out-of-order messages



request

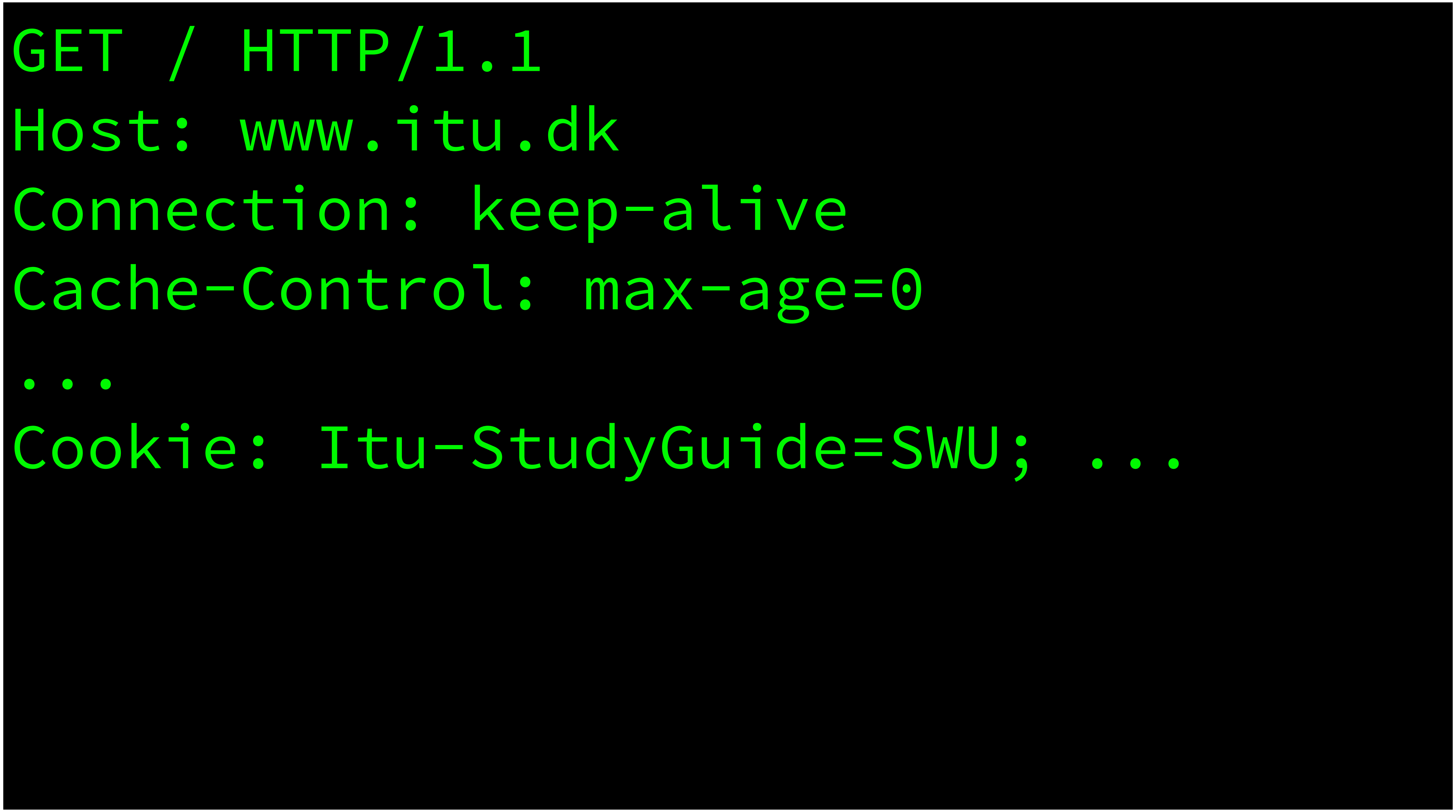
method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

reply

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

HTTP

Principle



```
GET / HTTP/1.1
Host: www.itu.dk
Connection: keep-alive
Cache-Control: max-age=0
...
Cookie: Itu-StudyGuide=SWU; ...
```

HTTP

Request

```
HTTP/1.1 200 OK
Date: Tue, 16 Sep 2014 12:07:10 GMT
Server: Microsoft-IIS/7.5
Cache-Control: no-cache, no-store
...
Connection: close


<!DOCTYPE html PUBLIC "-//W3C//DTD
XHTML 1.0 ...
```

HTTP

Request

HTTP & Resources

- A resource is an artifact with state, e.g. a document, an image, an airplane booking, a task on your to-do-list, ...
- It can be *created*, *read*, *updated* & *deleted* as supported directly by HTTP:

Application Task	HTTP Method
Create	POST: sending data
Read	GET: To retrieve a resource from specified URI
Update	PUT: To store a resource at specified URI
Delete	DELETE: To delete a resource

Safety & Idempotence

Application Task	HTTP Method
Create	POST: sending data
Read	GET: To retrieve a resource from specified URI
Update	PUT: To store a resource at specified URI
Delete	DELETE: To delete a resource

- GET (and HEAD) supposed to be *safe, i.e. no side-effects*
- GET, PUT, DELETE (,HEAD and OPTIONS) supposed to be *idempotent, i.e. no additional effect if repeated*

Remote Procedure Call (RPC)

- Transparently call remote procedures.
- What are the problems?
- Interface
- Failures

```
// 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();
};
```

Interface definition languages

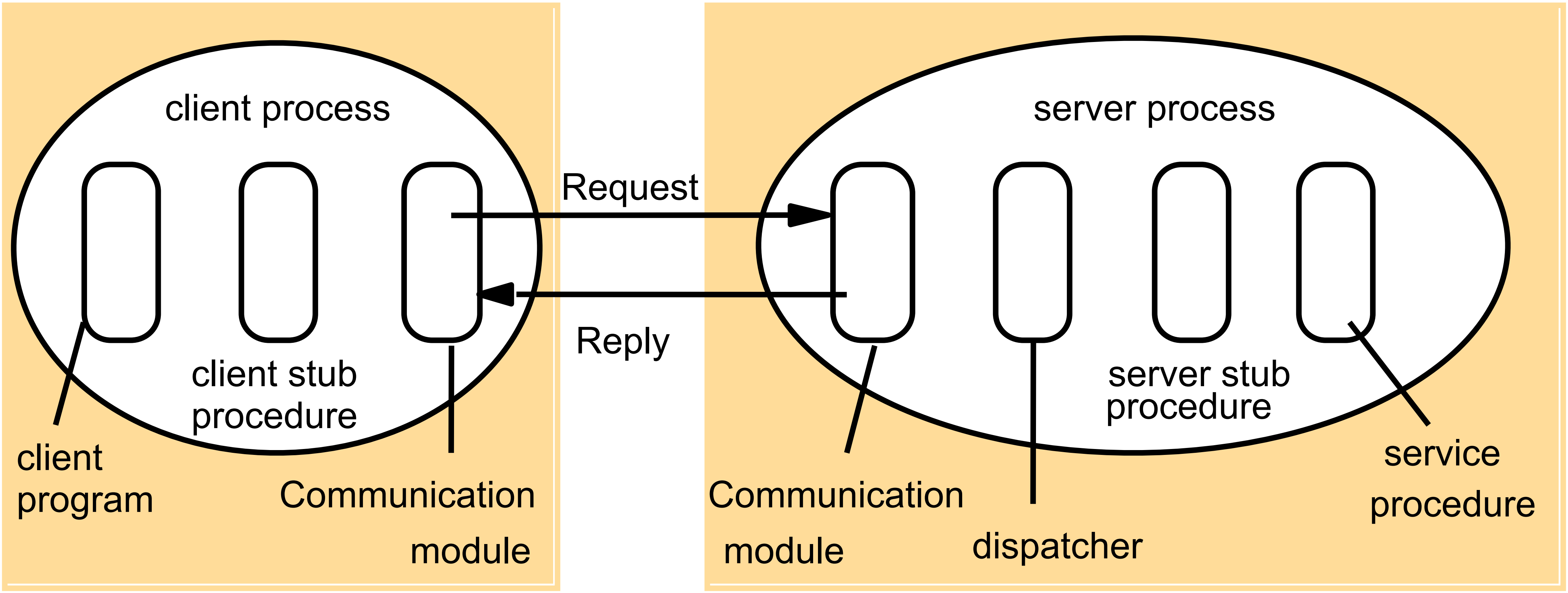
Here, CORBA.

RPC Goals

- Transparency through failure masking
- Can we have this? Can it be completely transparent?

RPC Semantics & failures

- Maybe semantics
  (Ie., “Maybe. Maybe not.”)
- At-least-once semantics
- At-most-once semantics



RPC implementation

Summary

- Request-reply
- HTTP
- RPC

Read on your own

- Lots, but especially:
- distributed garbage collection
- implementation details

Web services

APIs for the Web

What & why

- HTTP as request-reply mechanism

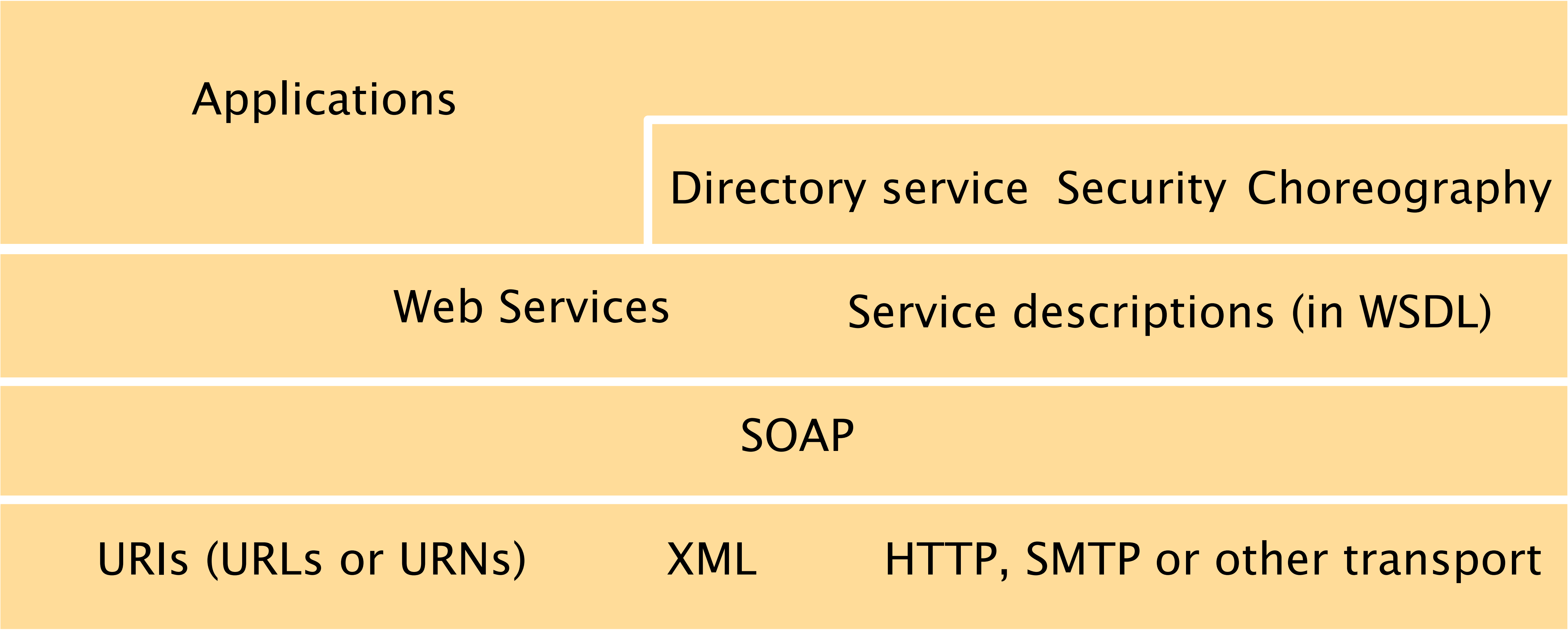
- URI = URL + URN
  (Identifier, Locator, Name)

- Operation descriptors

- Textual representations

- … basically, like XML: Fix conventions for the obvious.

Figure 9.1
Web services infrastructure and components



Applications

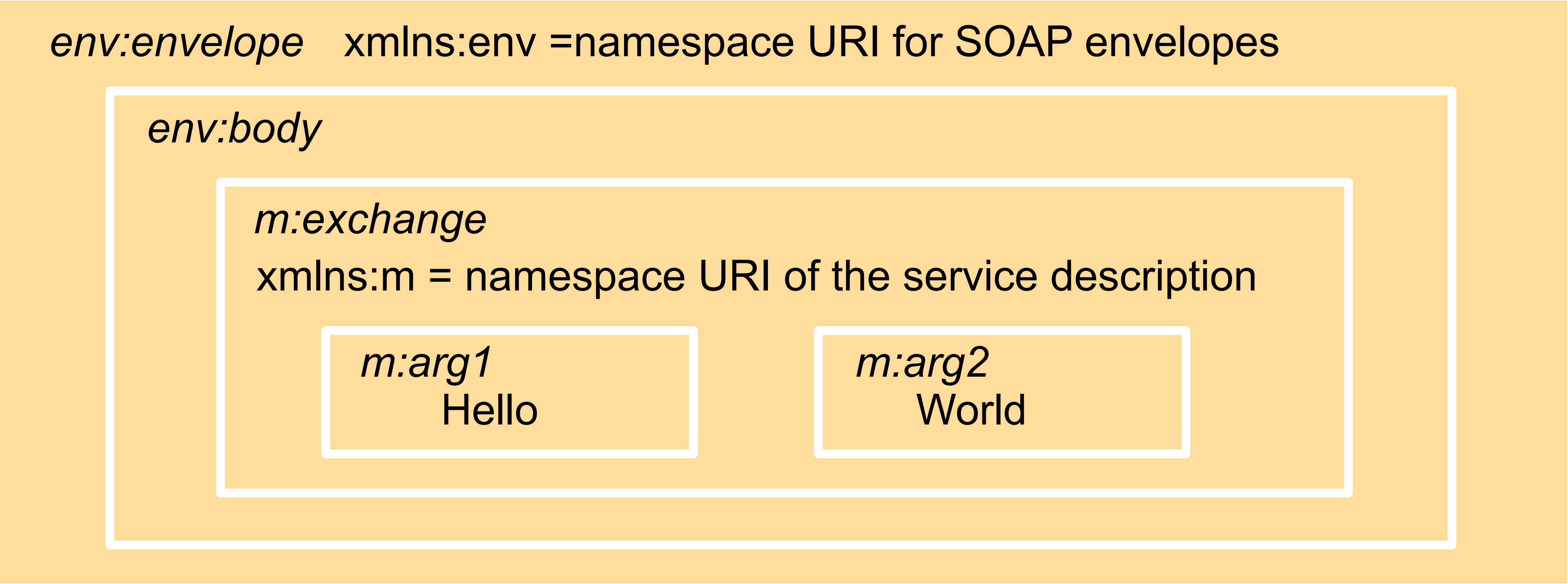
Directory service Security Choreography

Web Services Service descriptions (in WSDL)

SOAP

URIs (URLs or URNs) XML HTTP, SMTP or other transport

Figure 9.4
Example of a simple request without headers



env:envelope xmlns:env =namespace URI for SOAP envelopes

env:body

m:exchange

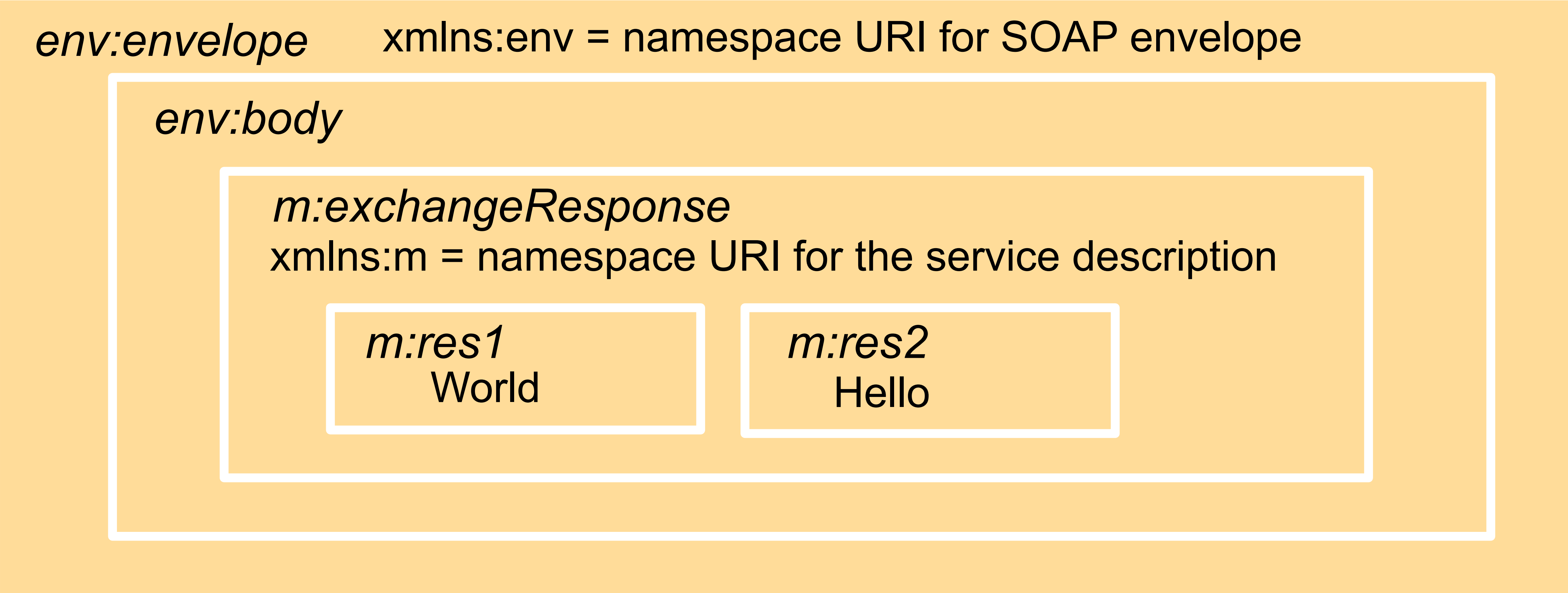
xmlns:m = namespace URI of the service description

m:arg1
Hello

m:arg2
World

In this figure and the next, each XML element is represented by a shaded box with its name in italic followed by any attributes and its content

Figure 9.5
Example of a reply corresponding to the request in Figure 9.4



env:envelope xmlns:env = namespace URI for SOAP envelope

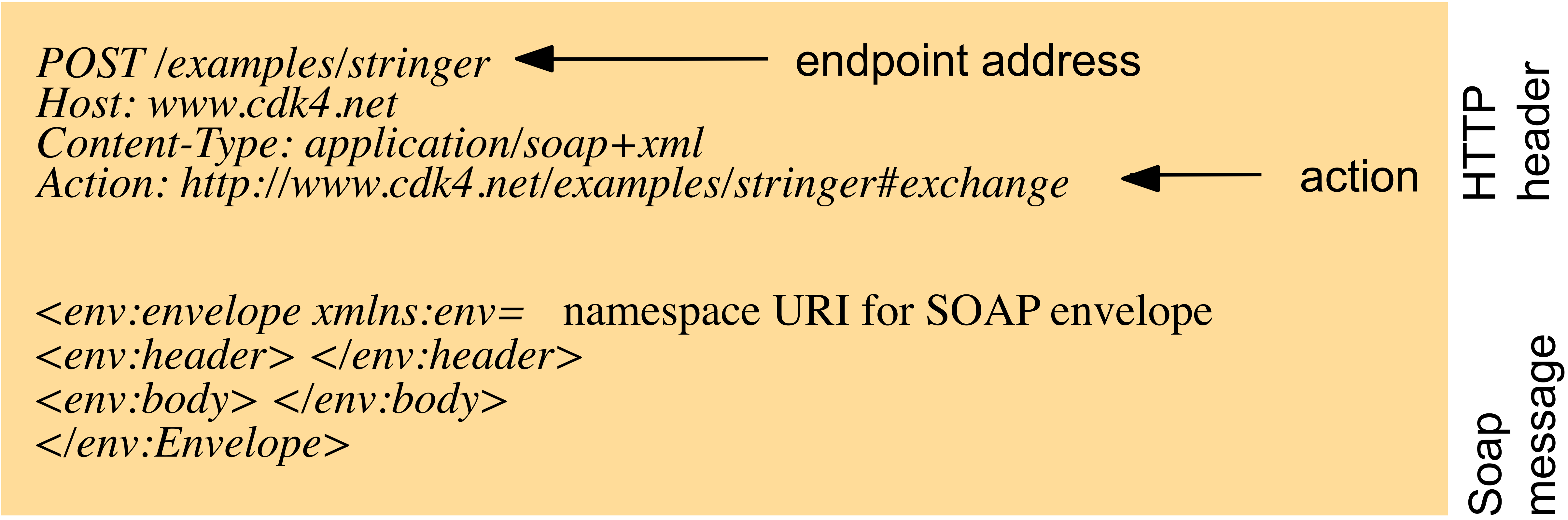
env:body

m:exchangeResponse
xmlns:m = namespace URI for the service description

m:res1
World

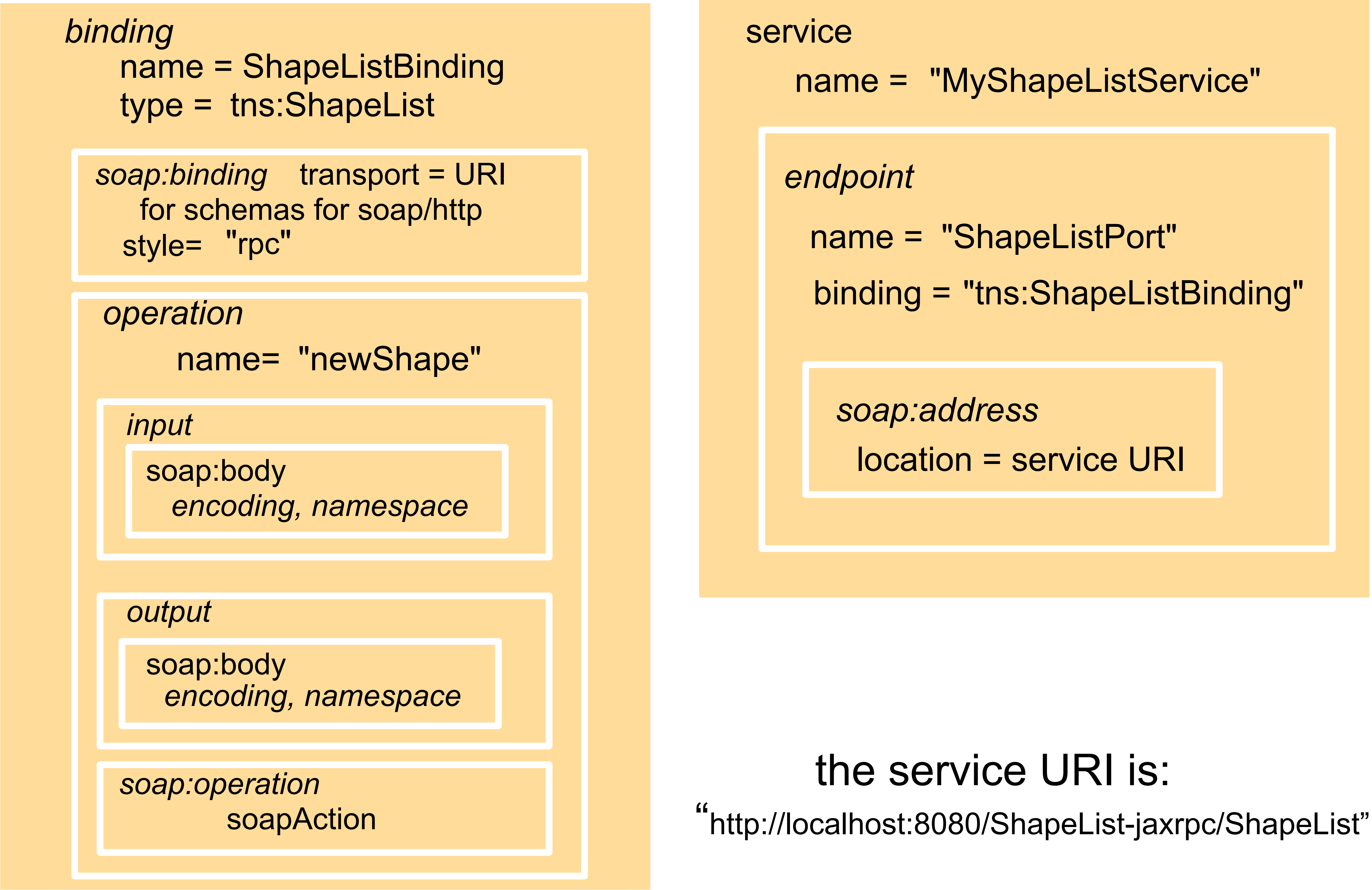
m:res2
Hello

Figure 9.6
Use of HTTP POST Request in SOAP client-server communication



```
POST /examples/stringer
Host: www.cdk4.net
Content-Type: application/soap+xml
Action: http://www.cdk4.net/examples/stringer#exchange


<env:envelope xmlns:env=   namespace URI for SOAP envelope
<env:header> </env:header>
<env:body> </env:body>
</env:Envelope>
```

Figure 9.14
SOAP binding and service definitions



binding
name = ShapeListBinding
type = tns:ShapeList

soap:binding transport = URI
for schemas for soap/http
style= "rpc"

operation
name= "newShape"

input
soap:body
encoding, namespace

output
soap:body
encoding, namespace

soap:operation
soapAction

service
name = "MyShapeListService"

endpoint
name = "ShapeListPort"
binding = "tns:ShapeListBinding"

soap:address
location = service URI

the service URI is:
“http://localhost:8080/ShapeList-jaxrpc/ShapeList”

REST

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml

<?xml version="1.0"?>
<user>
  <name>Robert</name>
</user>
```

REST: Creating

PUT

```
PUT /users/Robert HTTP/1.1
Host: myserver
Content-Type: application/xml

<?xml version="1.0"?>
<user>
  <name>Bob</name>
</user>
```

REST: Updating

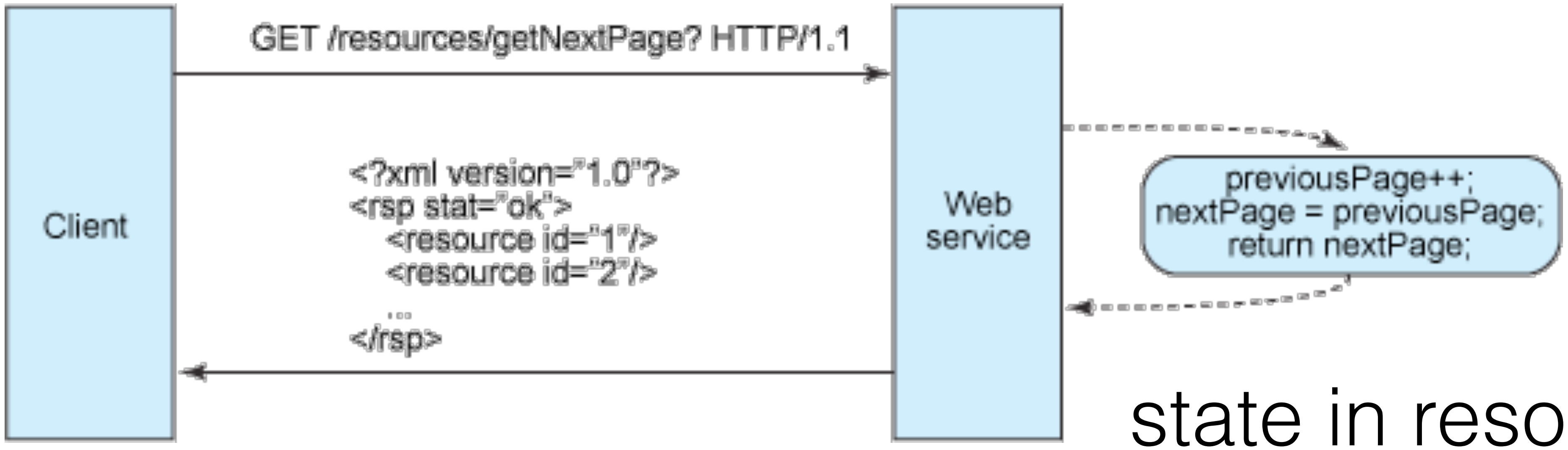
```
GET /users/Robert HTTP/1.1
Host: myserver
Accept: application/xml
```

REST: Retrieving

GET

What about state?

Stateful Design:



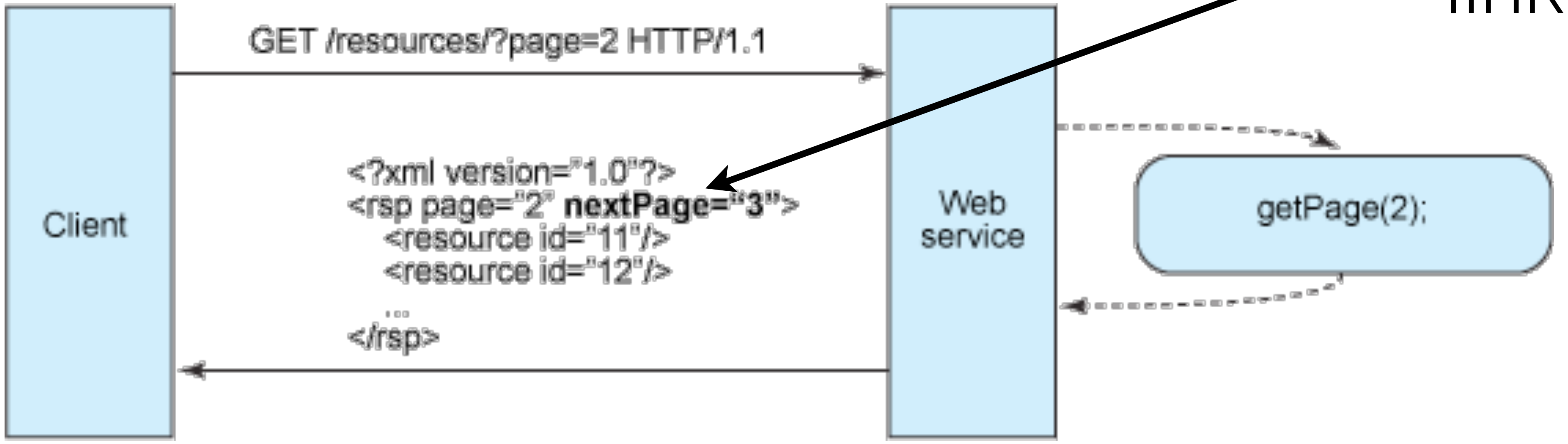
```
GET /resources/getNextPage? HTTP/1.1
```

```
<?xml version="1.0"?>
<rsp stat="ok">
   <resource id="1"/>
   <resource id="2"/>
   ...
</rsp>
```

```
previousPage++;
nextPage = previousPage;
return nextPage;
```

state in resource as a "link"

Stateless Design:



```
GET /resources/?page=2 HTTP/1.1
```

```
<?xml version="1.0"?>
<rsp page="2" nextPage="3">
   <resource id="11"/>
   <resource id="12"/>
   ...
</rsp>
```

```
getPage(2);
```

client send a complete request independent of past

source: Alex Rodriguez, RESTful Web services: The basics. http://www.ibm.com/developerworks/webservices/library/ws-restful/

Q: “What is the great joy of stateless services?”

A: “Ten thousand guests fed by a single grain of rice.”

Q: “And what is the great sorrow?”

A: “The great sorrow of what?”

http://thecodelesscode.com/case/96

Read on your own

- Details
- Fault handling
- WSDL
- Service discovery

Mini-project 2

Don’t give away the connection.

Questions?