جامعة الامير سلطان
PRINCE SULTAN UNIVERSITY

COMMUNICATION

Contd.(2)

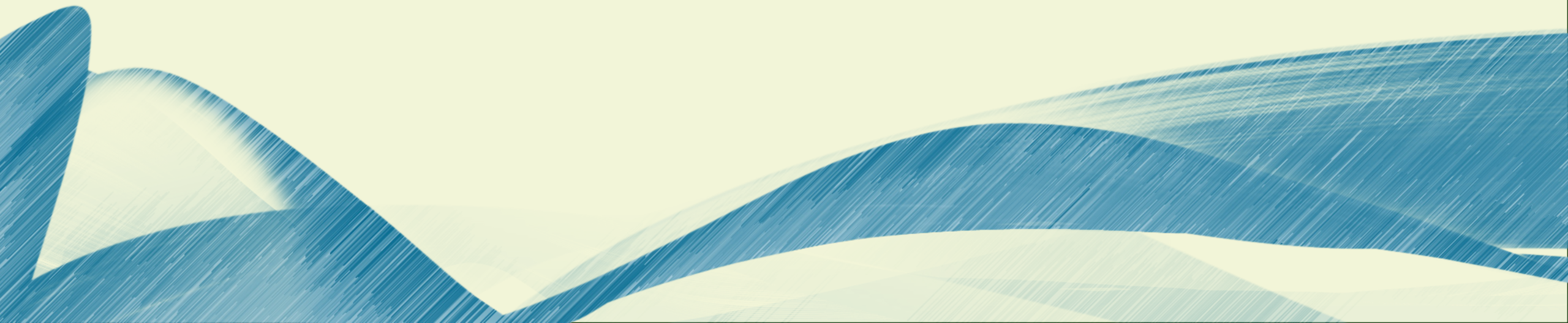
CS435 Distributed Systems

Basit Qureshi PhD, FHEA, SMIEEE, MACM
https://www.drbasit.org/

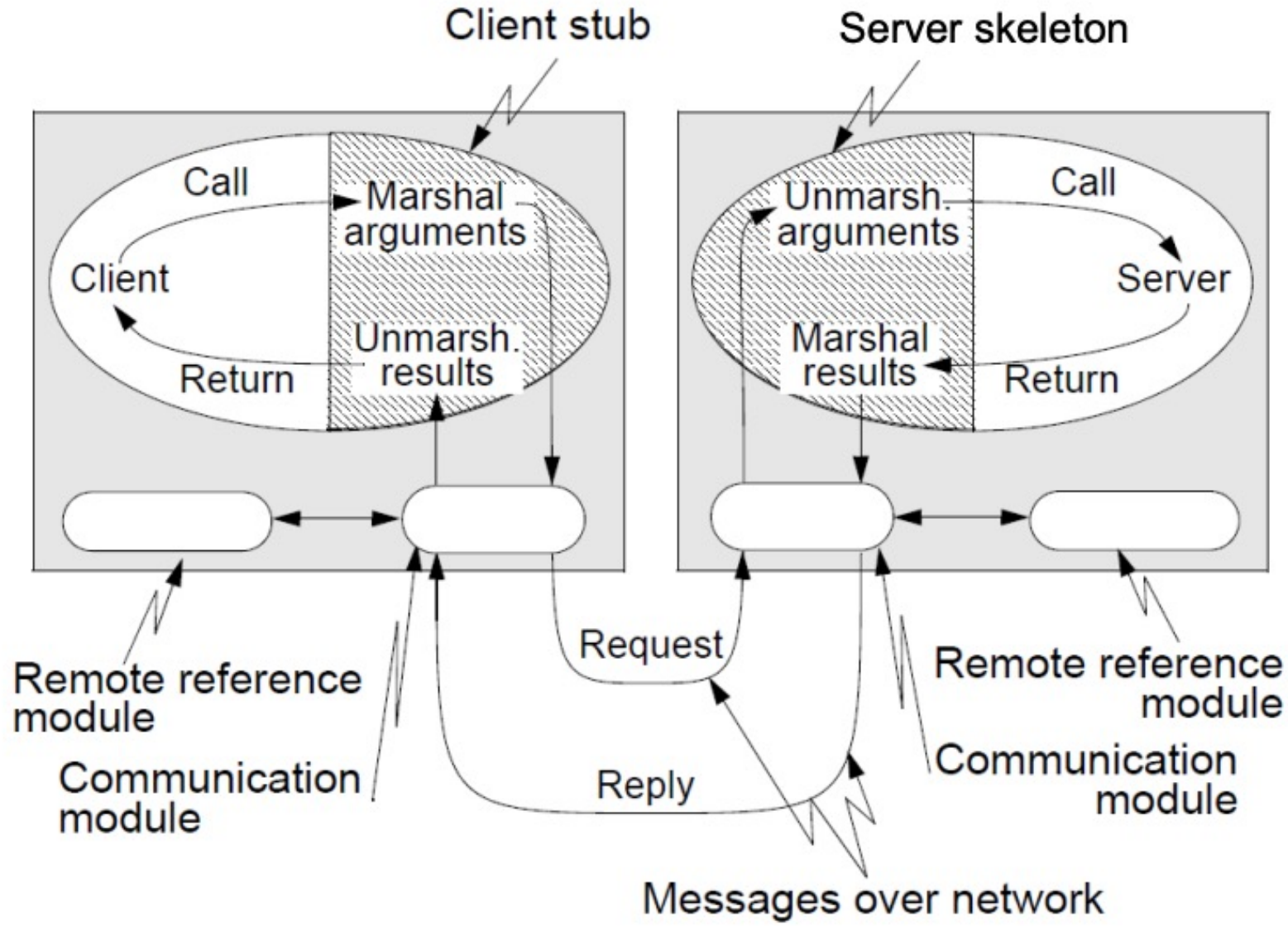
TOPICS

- RMI Semantics and Failures
- Direct vs Indirect / Group Communication
- Publish –subscribe systems
- Web-services

RMI SEMANTICS AND FAILURES



RMI SEMANTICS AND FAILURES



RMI SEMANTICS AND FAILURES

- If everything works OK, RMI behaves exactly like a local invocation. What if certain **failures** occur?

- Classes of failures that have to be handled by an RMI protocol:
  - Lost request message
  - Lost reply message
  - Server crash
  - Client crash
- We consider an omission failure model. This means:
  - Messages are either lost or received correctly.
  - Client or server processes either crash or execute correctly. After a crash, the server can possibly restart with or without loss of memory

RMI SEMANTICS AND FAILURES

- Lost Request Messages
  - The communication module starts a timer when sending the request
  - If the timer expires before a reply or acknowledgment comes back, the communication module sends the request message again.

PROBLEM

What if the request message was not truly lost (but, for example, the server is too slow) and the server receives it more than once?

RMI SEMANTICS AND FAILURES

- We must avoid that the server executes operations more than once.
- Messages have to be identified by an identifier and copies of the same message have to be filtered out:
  - If the duplicate arrives and the server has not yet sent the reply

    → simply send the reply.
  - If the duplicate arrives after the reply has been sent

    → the reply may have been lost or it did not arrive in time

RMI SEMANTICS AND FAILURES

- Lost Reply Message
  - The client can not distinguish the loss of a request from that of a reply; it simply resends the request because no answer has been received!
    - If the reply really got lost → when the duplicate request arrives at the server, it already has executed the operation once!
    - In order to resend the reply, the server may need to re-execute the operation in order to get the result.

Danger!??

RMI SEMANTICS AND FAILURES

- Lost Reply Message
  - Some operations can be executed more than once without any problem; they are called **idem-potent** operations

    → no danger with executing the duplicate request.
  - There are operations which cannot be executed repeatedly without changing the effect (e.g. transferring an amount of money between two accounts)

    → history can be used to avoid re-execution.
  - History (log): stores a record of reply messages that have been transmitted, together with the message identifier and the client which it has been sent to.

RMI SEMANTICS AND FAILURES

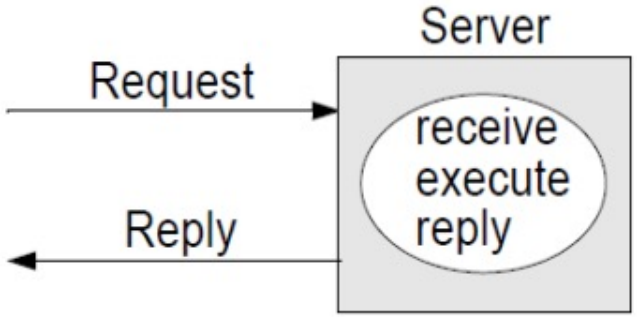
- **Solution!**
- **Exactly-once semantics** can be implemented in the case of lost (request or reply) messages if both **duplicate filtering** and **history** are provided and the message is resent until an answer arrives:
  - Eventually a reply arrives at the client and the call has been executed correctly - exactly one time.

What if the Server Crashes..!??

RMI SEMANTICS AND FAILURES

Server Crash!

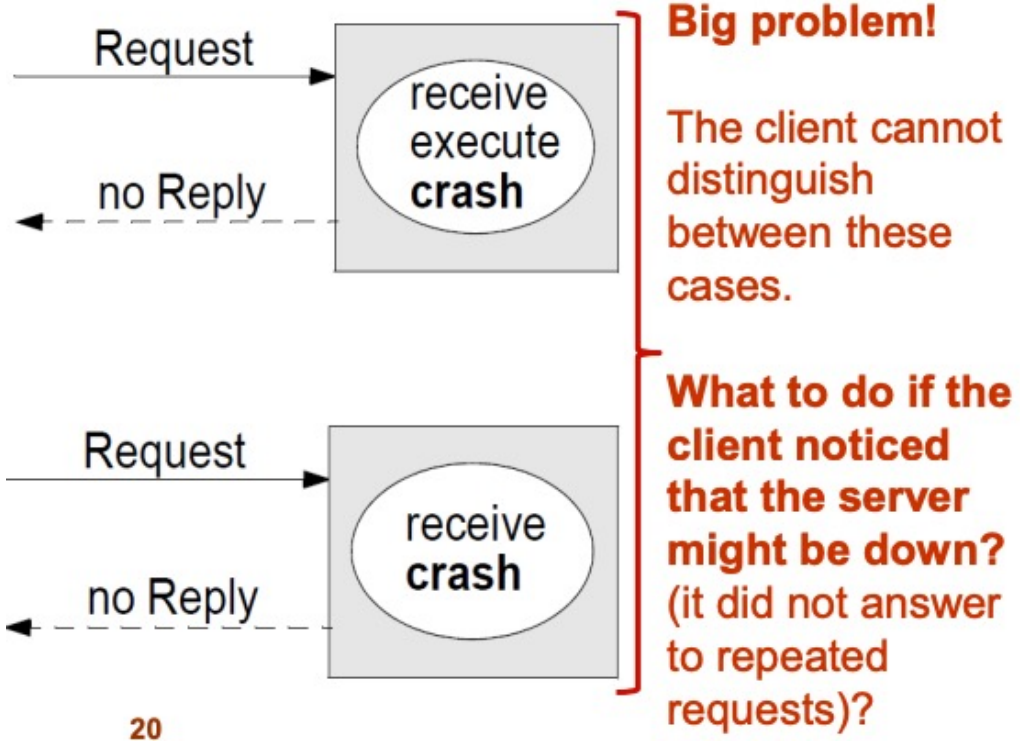
(a) The normal sequence:



(b) The server crashes ***after*** executing the operation, but ***before*** sending the reply:

- As result of the crash, the server lost memory and does not remember that it has executed the operation.

(c) The server crashes ***before*** executing the operation.



RMI SEMANTICS AND FAILURES

- **Server Crash**

Alternative 1: at-least-once semantics

▪ The client's communication module sends repeated requests and waits until the server reboots or it is rebound to a new machine; when it finally receives a reply, it forwards it to the client.

▪ When the client got an answer, the RMI has been carried out at least one time, but possibly more.

RMI SEMANTICS AND FAILURES

- **Server Crash**

Alternative 2: at-most-once semantics

▪ The client's communication module gives up and immediately reports the failure to the client (e.g., by raising an exception).

▪ If the client got an answer, the RMI has been executed exactly once.

▪ If the client got a failure message, the RMI has been carried out at most one time, but possibly not at all.

RMI SEMANTICS AND FAILURES

- **Server Crash**

Alternative 3: exactly-once semantics

- This is what we would like to have (and what we could achieve for lost messages): the RMI has been carried out exactly one time.
- **However, this cannot be guaranteed, in general, for server crashes**

RMI SEMANTICS AND FAILURES

- **Client Crash**

The client sends a request to a server and crashes before the server replies.

▪ The computation which is active in the server becomes an **orphan** - a computation nobody is waiting for.

Problems:

- Waste of server CPU time
- Locked resources (files, peripherals, etc.)
- If the client reboots and repeats the RMI, confusion can be created.

The solution is based on identifying and killing the orphans

RMI SEMANTICS AND FAILURES

Summary

- If the problem of errors is ignored, **maybe-semantics** is achieved for RMI:
  - The client, in general, does not know if the remote method has been executed once, several times, or not at all.
- If server crashes can be excluded, **exactly-once semantics** is achieved for RMI:
  - Resending requests, filtering out duplicates, and using history.

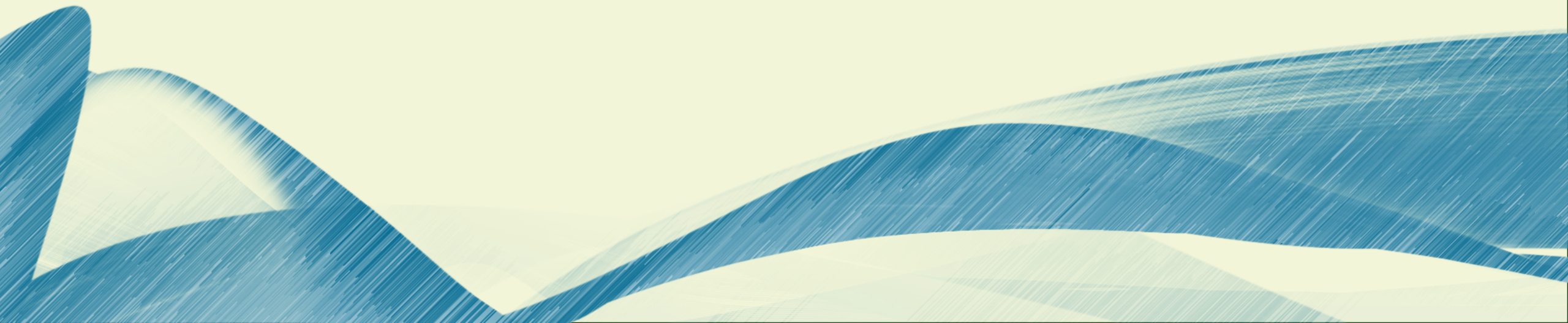
- If server crashes **with loss of memory** are considered, only **at-least-once** and **at-most-once** semantics are achievable in the best case.

RMI SEMANTICS AND FAILURES

Summary (contd)

- In practical applications, servers can survive crashes without loss of memory.
  - **Transaction-based sophisticated commitment protocols** are implemented in distributed systems to achieve this goal.
  - In such systems, **history** can be used and **duplicates** can be filtered out after restart of the server
  - The client repeats sending requests without being in danger operations to be executed more than once:
    - If **no answer** is received after a certain amount of tries, the client is notified, so it knows that the method has been executed at most once or not at all.
    - If **an answer** is received, it is forwarded to the client, which knows that the method has been executed exactly one time.
- RMI implementation and error handling differs between systems. Sometimes several semantics are implemented, among which the user is allowed to select.

DIRECT VS INDIRECT / GROUP COMMUNICATION



DIRECT VS INDIRECT COMMUNICATION

- The communication primitives studied so far are based on **direct coupling between sender and receiver:** the sender has a reference/pointer to the receiver and specifies it as an argument of the communication primitive.
- The sender writes something like:

```
...
send (request) to server_reference;
...
```

→ Very Rigid!

DIRECT VS INDIRECT COMMUNICATION

An alternative: **Indirect communication**

- No direct coupling between sender and receiver(s).
- Communication is performed via an **intermediary**.

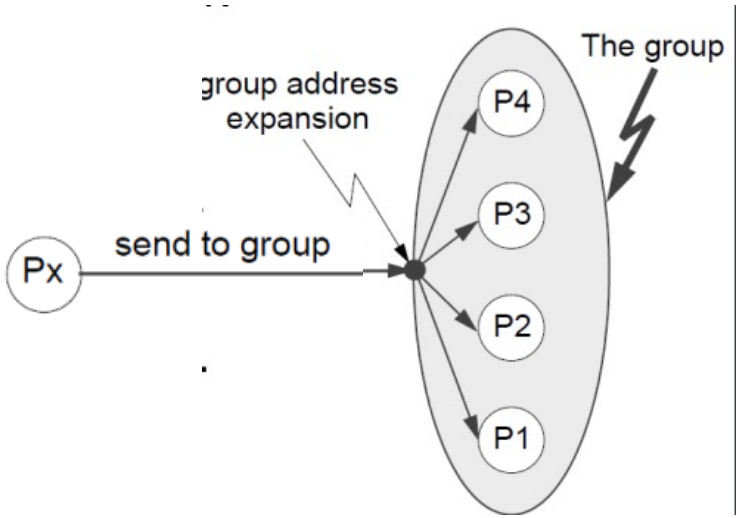
- **Space decoupling:**
  sender does not know the identity of receiver(s).
- **Time decoupling:**
  sender and receiver(s) have independent lifetimes: they do not need to exist at the same time.
- We look at two examples:
  1. Group communication
  2. Publish-subscribe systems

DIRECT VS INDIRECT COMMUNICATION

Group Communication

- The assumption with client-server communication and RMI (RPC) is that two parties are involved: *the client and the server.*
- Sometimes communication involves multiple processes, not only two.
  - A (simple) solution is to perform separate message passing operations or RMIs to each receiver.
- With group communication, a message can be sent to a group and then it is delivered to all members of the group

  → multiple receivers in one operation.



DIRECT VS INDIRECT COMMUNICATION

Group Communication: Why do we need it?

- **Special applications:**
  - interest-groups, mail-lists, etc.
- **Fault tolerance based on replication:**
  - A request is sent to several servers which all execute the same operation (if one fails, the client still will be served).
- **Locating a service or object in a distributed system:**
  - The client sends a message to all machines, but only the one (or those) which holds the server/object responds.
- **Replicated data (for reliability or performance):**
  - whenever the data changes, the new value has to be sent to all processes managing replicas.

DIRECT VS INDIRECT COMMUNICATION

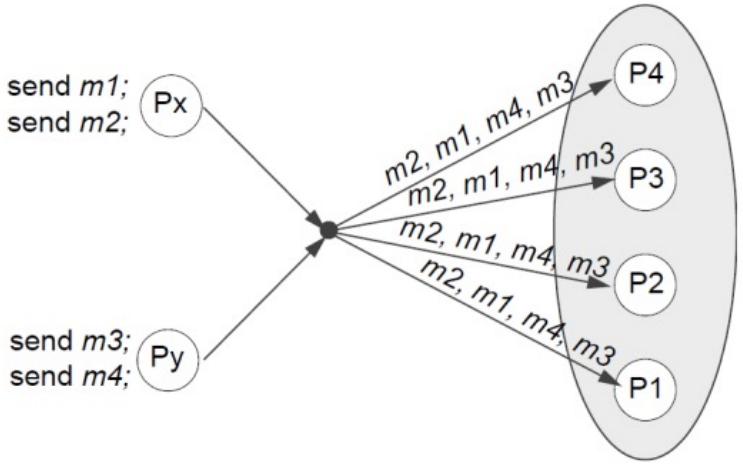
Group Communication:

- **Group membership management:**
  - maintains the view of group membership, considering members joining, leaving, or failing.
- **Services provided** by group membership management:
  - **Group membership changes:**
    - create/destroy process groups;
    - add/withdraw processes to/from group.
  - **Failure detection:**
    - Detects processes that crash or become unavailable (due to e.g. communication failure);
    - Excludes processes from membership if crashed or unavailable.
  - **Notification**:
    - Notifies members of events, e.g., processes joining/leaving group.
  - **Group address expansion:**
    - Processes sending to a group specify the group identifier;
    - **Address expansion** provides the actual addresses for the multicast operation delivering the message to each group members.

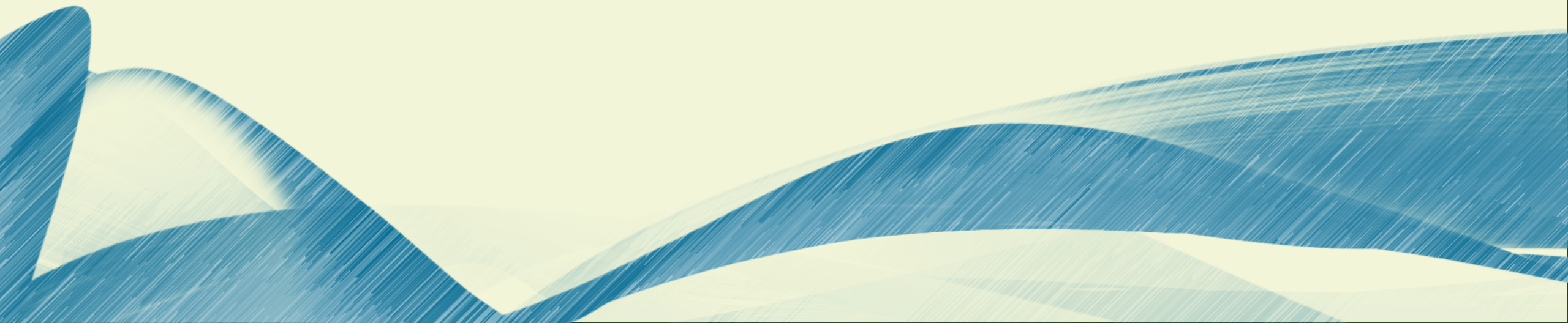
DIRECT VS INDIRECT COMMUNICATION

Group Communication: Essential features:

- **Atomicity** (all-or-nothing):
  - when a message is sent to a group, it will either arrive correctly at all members of the group or at none of them.
- **Ordering**
  - **FIFO-ordering:** Messages originating from a given sender are delivered in the order they have been sent, to all members of the group.
  - **Total-ordering**: When several messages, from different senders, are sent to a group, the messages reach all the members of the group in the same order



PUBLISH SUBSCRIBE SYSTEMS

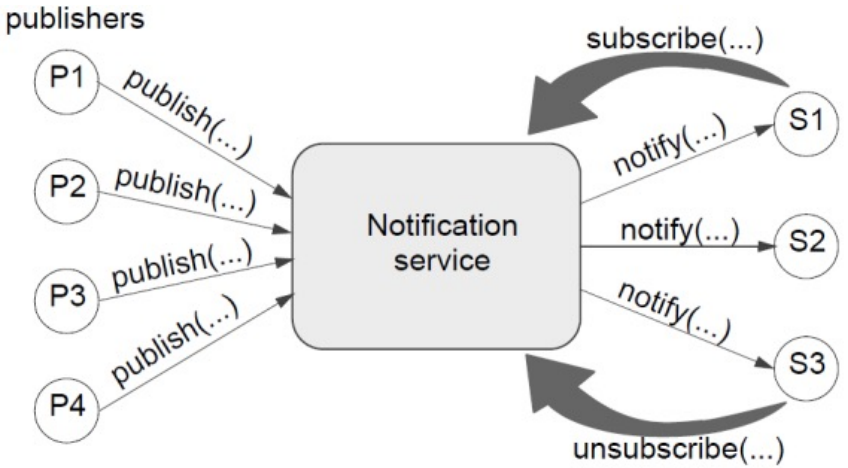


PUBLISH-SUBSCRIBE SYSTEMS

- The general objective of publish-subscribe systems is ***to let information propagate from publishers to interested subscribers, in an anonymous, decoupled fashion.***
- **Publishers** publish events.
- **Subscribers** subscribe to and receive the events they are interested in.
- Subscribers are not directly targeted from publishers but indirectly via the notification service →
  - Subscribers express their interest by issuing subscriptions for specific notifications, independently from the publishers that produces them;
  - They are asynchronously notified for all notifications, submitted by any publisher, that match their subscription.

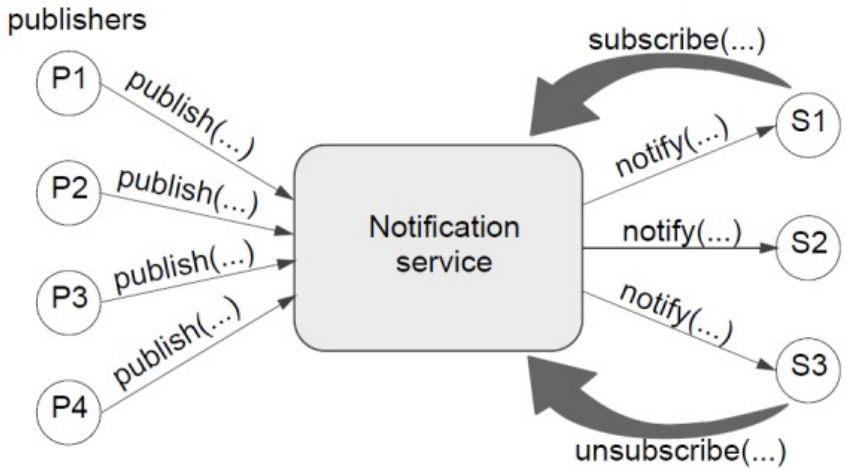
PUBLISH-SUBSCRIBE SYSTEMS

- **Notification Service**: is a propagation mechanism that acts as a **logical intermediary ("broker")** between publishers and subscribers, to avoid each publisher to have to know all the subscriptions for each possible subscriber.
  - Both publishers and subscribers communicate only with a single entity, the notification service, which
    - stores the **subscriptions** associated with each subscriber;
    - receives all the **notifications** from publishers;
    - **dispatches** the notifications to the correct subscribers



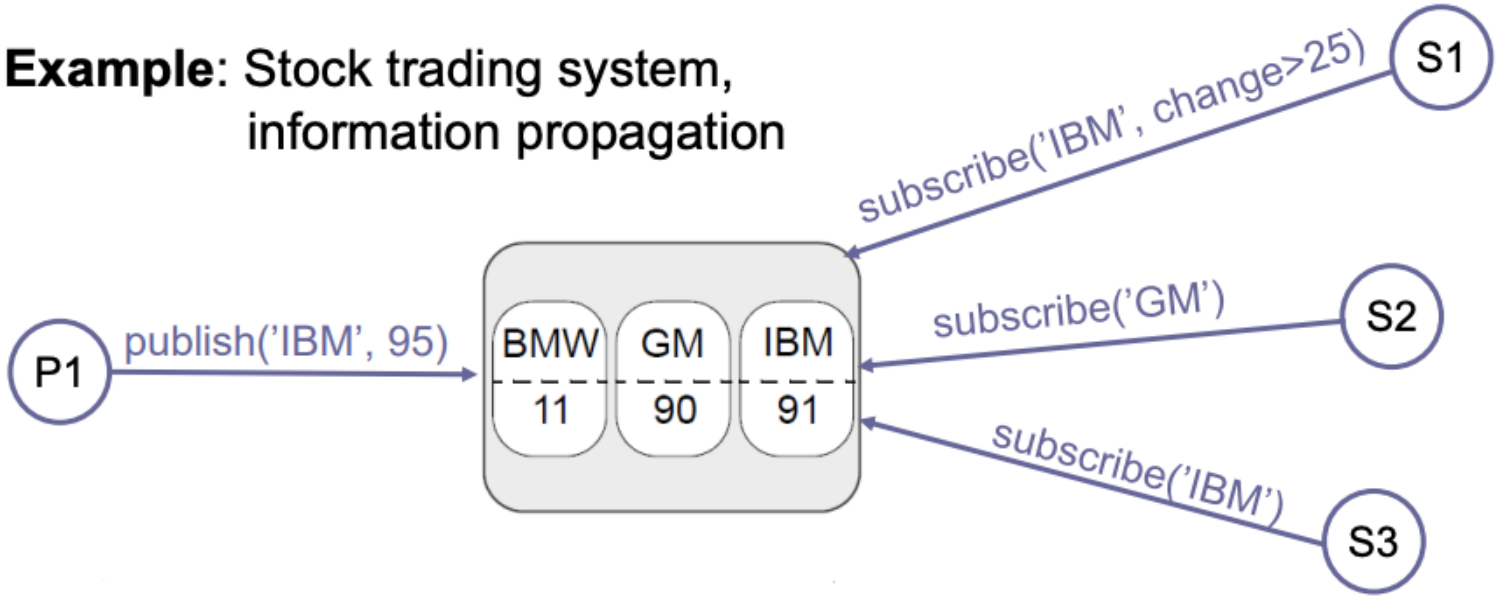
PUBLISH-SUBSCRIBE SYSTEMS

- A publisher submits a piece of information by executing the **publish()** operation on the notification service.
- The notification service dispatches a piece of information to a subscriber by executing the **notify()** on it.
  - A publisher produces an event (publication), while the notification service issues the corresponding notification on interested subscribers.



PUBLISH-SUBSCRIBE SYSTEMS

- **Example**: Stock trading system, information propagation



1. S1 has subscribed to 'IBM', with a filter indicating that it should be notified only if the stock increases by at least 25;

2. S2 and S3 have subscribed to 'GM' and 'IBM' respectively, without filter.

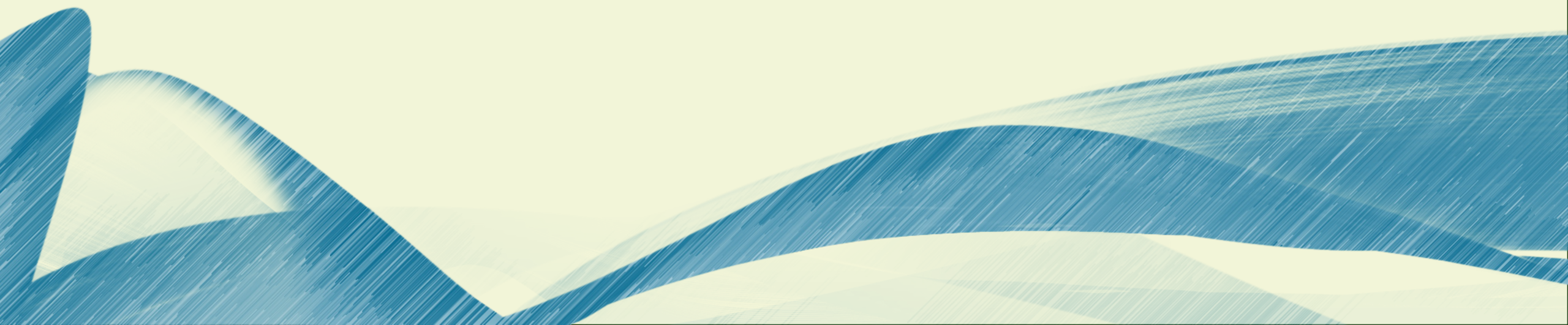
3. P1 is publishing the new value of 'IBM'. Who is notified?

4. S1 is not notified because its filter is not satisfied;
S2 is not notified because it's not interested in 'IBM';
S3 is notified.

35

NOTE: Slide taken from: Christoph Kessler IDA Linköping University Sweden

WEB SERVICES



WEB SERVICES

The early web (1990s)

- The early web was based on the concept of *hypertext*, allowing researchers to link documents together.
  - HTML (Hypertext Markup Language) was introduced as a means to format and structure this content.
  - Websites were collections of HTML documents hosted on servers, and every user received the same content.

There was no dynamic content generation or personalization.

- **Web browsers**, such as the early versions of Netscape Navigator and Internet Explorer, were developed to interpret HTML and present formatted content to users.
  - They acted as clients requesting static pages from servers.

WEB SERVICES

The early web (1990s)

- The **Common Gateway Interface** (CGI) was one of the early methods used to provide dynamic web content.
  - CGI scripts, enabled servers to capture requests and generate dynamic content, providing a basic level of interactivity.
- Support for **Java applets** was introduced into the web browser.
  - Applications written in Java and embedded within web pages and enabled interactive content on web pages.

[2000+] Components in the web browser evolved to create richer interactive pages

- **Cascading Style Sheets** (**CSS**) Impressive HTML displays
- **JavaScript** a scripting language built into a browser; code embedded in a web page interact with user input.
- **Document Object Model** (**DOM**) All aspects of the page are organized into objects.
- **Asynchronous JavaScript and XML (AJAX)**: highly interactive websites where page content can change dynamically based on updates from a server. (Google Maps)

WEB SERVICES

Why not RPC?

- RPCs work with **arbitrary ports** controlled by the *server OS*. It would be a nightmare for a webserver administrator to manage ports for each connection. Client need to be access resources on a singular port behind a firewall.
- Most RPCs are **technology-locked**
  - Sun-RPC doesnot work on IBM Linux
  - DCE-RPC only generate C code
  - Microsoft services work only in Microsoft Ecosystem (DCOM, .NET etc)
  - Cross-platform solutions (CORBA) not always interoperable
- Some services needs publish-subscribe (pub/sub) interface.
- RPCs were designed with **Local Area Networks** in mind. Never designed for unlimited access services.
- RPC does not store **state of the system**. This is required for web sessions!

WEB SERVICES

Principles of webservices

- **Web services** are a set of protocols by which services can be published, discovered, and used over the Internet in a technology-neutral form.
  - This means that they are designed to be language and architecture independent.
  - Applications will typically invoke multiple remote services across different systems, sometimes offered by different organizations.
- Use HTTP over TCP/IP for transport
- **Be platform agnostic**: Web services aimed to be platform and language neutral.
- **Standard:** Services would operate over internet protocols and use a well-defined schema for messaging.
- Use text-based payloads, called **documents**, marshalling all data into formats such as XML or JSON.
- **Tolerate high latency.** Servers are likely not to be on a local area network and may be slow to respond either due to their own load or due to network latency.
- **Tolerate a large number of clients**

Functionally, you can do anything with web services that you can with distributed objects (RPC).

WEB SERVICES

Service Oriented Architecture

Service-Oriented Architecture, or **SOA**, is a software architectural pattern focused on the distribution and operation of loosely-coupled services.

- Constructs Software Components (Services)
- Services can be re-used across various applications

Key Ideas:

- **Unassociated**: No service depends on another service; they are all mutually independent.
- **Loose Coupling**: Neither service needs to know about the internal structure of other services.
- **Interoperability:** Allows services to communicate using standard protocols, e.g. SOAP.
- **Discoverability**: Facilitates the finding and understanding of services using directories or registries.
- **Reusability**: Encourages building services for multiple contexts, promoting efficiency.

WEB SERVICES

Simple Object Access Protocol (SOAP)

An XML-based protocol that allowed programs on disparate systems to communicate over HTTP.

- SOAP invocations are always XML messages that are usually sent via an HTTP protocol. May also send a SOAP message via email and SMTP (Simple Mail Transport Protocol).
- **WSDL (Web Services Description Language)** allows clients to discover the methods provided by the web service.
  - SOAP services can be described via the Web Services Description Language (WSDL) is an XML-based document that describes a specific web service.
  - It serves as an interface definition and defines all the names, operations, parameters, destination, and format of requests.

WEB SERVICES

Example: **A SOAP message to add two values**

- Envelope: The outermost element that encapsulates the entire SOAP message.
- Header (optional): Contains any additional information, such as authentication credentials / message routing details.
- Body: Contains the actual request data.

WEB SERVICES

Example: **A SOAP message to add two values**

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:ns="http://example.com">

    <soap:Header/>
    <soap:Body>
        <ns:AddRequest>
            <ns:Value1>5</ns:Value1>
            <ns:Value2>10</ns:Value2>
        </ns:AddRequest>
    </soap:Body>
</soap:Envelope>
```

A Client sends AddRequest with two values Value1 and Value2

WEB SERVICES

Example: **A SOAP message to add two values**

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:ns="http://example.com">
    <soap:Header/>
    <soap:Body>
        <ns:AddResponse>
            <ns:Sum>15</ns:Sum>
        </ns:AddResponse>
    </soap:Body>
</soap:Envelope>
```

Server Receives two values, adds and returns the Sum; here it is 15

WEB SERVICES

Microservices

A microservices architecture represents a modern approach to constructing software systems, emphasizing modularity and scalability. Microservices decompose a software application into smaller, independently operable services, each tailored for a specific function.

Key Ideas:

- **Fine-Grained**: Focuses on doing one specific thing efficiently.
- **Independence**: Enables each service to be developed, tested, deployed, and scaled autonomously.
- **Decentralized Data Management**: Each service usually manages its data, promoting loose coupling.
- **Statelessness**: Prefers not to maintain client-specific session information between requests.

WEB SERVICES

RESTful web services

REST stands for **Representational State Transfer**. RESTful services are built on top of the HTTP protocol and utilize its methods and features to provide interoperability between different systems on the internet.

The **URI** (**Uniform Resource Identifier**, usually a URL) incorporates the request and list of parameters. The HTTP protocol itself defines the core operations (the verbs):

- **POST**: create something
- **PUT**: create or replace something
- **PATCH**: update a part of something
- **GET**: read something
- **DELETE**: delete something

The body of the message will contain the document, which will contain data for the operation

WEB SERVICES

Example: **A RESTful request and response to add two values**

```
POST /add HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "num1": 5,
    "num2": 10
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "result": 15
}
```

This example assumes that a server listening at example.com that handles POST requests to the /add endpoint. The server would parse the JSON request body, extract num1 and num2, add them together, and then respond with the result in JSON format.

WEB SERVICES

Server implementation using Node.js

```
// Import required modules
const express = require('express');
const bodyParser = require('body-parser');

// Initialize Express app
const app = express();
const port = 3000;

// Middleware for parsing JSON body
app.use(bodyParser.json());

// Define POST route for adding two numbers
app.post('/add', (req, res) => {
    // Extract numbers from request body
    const { num1, num2 } = req.body;
```

WEB SERVICES

Server implementation using Node.js

```
    // Check if both numbers are provided
    if (num1 === undefined || num2 === undefined) {
        return res.status(400).json({ error: 'Both numbers are required' });
    }

    // Check if the provided inputs are numbers
    if (isNaN(num1) || isNaN(num2)) {
        return res.status(400).json({ error: 'Invalid input.' });
    }

    // Calculate the sum
    const sum = parseFloat(num1) + parseFloat(num2);

    // Send the result as JSON response
    res.json({ result: sum });
});
```

WEB SERVICES

Server implementation using Node.js

```
// Start the server
app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

- The server will be running on http://localhost:3000.
- Send POST requests to http://localhost:3000/add with a JSON body containing num1 and num2,
- Server will respond with the sum of the two numbers

```
{"result":15}
```

WEB SERVICES

Server implementation using Flask (python)

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/add', methods=['POST'])
def add_numbers():
    data = request.get_json()
    num1 = data['num1']
    num2 = data['num2']
    result = num1 + num2
    return jsonify({'result': result})

if __name__ == '__main__':
    app.run(debug=True)
```

This Flask app sets up a server with a single /add endpoint that accepts POST requests.
It parses the JSON request body, adds the numbers, and returns the result in JSON format.

COMPARING SOA & MICROSERVICES

SOA Architecture

- **Granularity**: SOA typically delivers coarser-grained services.
- **Communication**: SOA, being older, would often lean toward the use of SOAP.
- **Data Management**: SOA services share a common database.
- **Deployment**: SOA may not offer this.
- **SOAP** defines a complete messaging protocol, detailing the definition and representation of data types and the structure of messages
- **SOAP** allows anyone with a WSDL file to be able to create a valid interface to a service

Microservices (REST)

- More fine-grained.
- Tend to favor RESTful APIs and lightweight messaging.
- Database-per-service model in microservices.
- A strong emphasis on independent deployment of each service.
- **REST** is based on standard HTTP methods (like GET, POST, PUT, DELETE) and status codes.
- **JSON** is typically less verbose than XML, leading to faster parsing and smaller message sizes.
- Responses could be labeled as cacheable or non-cacheable, optimizing performance by reducing the need for some client-server interactions.

SUMMARY

- RMI Semantics and Failures
- Direct vs Indirect / Group Communication
- Publish –subscribe systems
- Web-services