Distributed Systems

Lectured by Emil C Lupu

Typed by Aris Zhu Yi Qing

April 19, 2022

Contents

1 Characteristics

1.1 Distribution Transparencies

Realize a coherent system by *hiding distribution* from the user where possible.

- **Access**: uniform access whether local or remote
- **Location**: access without knowledge of location
- **Concurrency**: sharing without interference (requires synchronization)
- **Replication**: hides use of redundancy (e.g. for fault tolerance)
- **Failure**: conceal failures by replication or recovery
- **Migration**: hides migration of components (e.g. for load balancing)
- **Performance**: hide performance variations (e.g. through use of scheduling and reconfiguration)
- **Scaling**: permits expansion by adding more resources (e.g. cloud)

1.2 Challenges

- **Heterogeneity**: different OS, data representation, implementations, etc.
- **Openness**: need to define *interfaces* for components to easily scale up systems
- **Security**: control access to preserve integrity and confidentiality
- **Concurrency**: inconsistencies may arise with interleaving requests
- **Failure handling**: transient/permanent failures could occur at any time. It is difficult detect them and to maintain consistency.
- **Scalability**: size of the system makes it difficult to maintain information about *system state*.

1.3 Wrong Assumptions

- The network is reliable, secure & homogeneous.
- The topology does not change.
- The latency is zero.
- The bandwidth is infinite.
- Transport cost is zero.
- There is one administrator.

1.4 Terminology

- **Client**: an entity initiating an interation
- **Server**: a componenet responds to interactions usually implemented as a process
- **Service**: a componenet of a computer system that manages a collection of resources and presents their functionality to users.
- **Middleware**: software layer between the application and the OS masking the heterogeneity of the underlying system.

2 Architecture

2.1 Layered architecture

- e.g. Network stack. Control flows downwards, results flow upwards.

+ framework is simple and easy to learn and implement
+ reduced dependency due to layer separation
+ testing is easier with such modularity
+ cost overheads are fairly low

- scalability is difficult due to fixed framework structure
- difficult to maintain, since a change in a single layer can affect the entire system because it operates as a single unit
- parallel processing is not possible

2.2 Object-based and service-oriented architectures

- e.g. RMI

+ reusability, easy maintainability and greater reliability due to modularity
+ improved scalability and availability: multiple instances of a single service can run on different servers at the same time.

- Increased overhead: Service interactions require validations of inputs, thereby increasing the response time and machine load, and reducing the overall performance

2.3 Message-based architectures

	Temporally coupled	Temporally decoupoled
Referentially coupled	Direct process messaging	Messaging via mailbox
Referentially decoupled	Event-based (publish-subscribe)	Shared data spaces

- *Referentially coupled*: processes name sender/receiver in their communication.
- *Temporally coupled*: both sender and receiver need to be up and running.

2.4 Peer-to-peer

- structured: Each node is indexed so that the location is known, and messages are routed according to the topology.
- unstructured: *flooding* or *random walks* or both.

+ no server needed since individual workstations are used to access files
+ resilient to computer failures, since it does not disrupt any other part of the network
+ very scalable

- poor performance with larger networks since each computer is being accessed by other users
- no central file system, hard to look up or backup
- ensuring that viruses are not introduced into the network is the responsibility of each individual user.
- There is no security other than assigning permissions.

3 Message-passing and IPC

- Asynchronous send: sender continues its execution once the message has been copied out of its address space
  + mostly used with *blocked receive*
  + underlying system must provide buffering for receiving messages independently of receiver processes
  + *loose* coupling: sender does not know when message will be received, does not suspend execution until the message has been received
  - *Buffer exhaustion* (no flow control)
  - formal verification is more difficult, as need to account for the state of the buffers
- Synchronous send: *blocked send*, where the sender is held up until actual receipt of the message by the destination.
  + usually used with blocking receive, where receiver execution is suspended until a message is received.
  + synchronization between sender and receiver
  + generallyl easier to formally reason about synchronous systems
  - what if no receivers? message loss?
  - No multi-destination, requiring synchronization with all receivers.
  - implementation more complicated
  - The underlying communication service is expected to be *reliable*, i.e. to guarantee in order message delivery.

- Asynchronous receive: process continues execution if there are no messages. hardly provided as primitives
- Blocked receive: the destination process blocks if no message is available, and receives it into a target variable when available.
- Please check the coursework for how UDP client/server is implemented in Java, e.g. how datagram, socket, port are used.

4 Complex Data Representation

4.1 Definition

- **Marshalling** takes a collection of data items and transform them in a format suitable for transmission.
- **Unmarshalling** reconstitutes the data values and data structures from the bytes received.

4.2 Encoding structures

Constructed types need to be flattened for transfer.

```
struct Person {
        string Name;
        string place;
        unsigned long year;
}
```

e.g., {“Smith”, “London”, 1984}

Alignment on multiples of 4 bytes. Variable length structures are “padded” to preserve the alignment.

index	4 byte	
0-3	5	*length*
4-7	“Smit”	
7-11	“h___”	
12-15	6	*length*
16-19	”Lond”	
20-23	”on__”	
24-27	1984	*unsigned long*

4.3 Composed structures

```
struct rec {
        int a;
        boolean b;
};
struct form {
        int x;
        float y;
        rec z  [ 3];  /* assume 3 elements */
};
form obj = (5, 23.75, 10, true, 5, false, 7, true)
```

can be “flattened” for transfer:
where I = int, F = float, B = boolean

$$\underset{x}{\mathbf{I}}\ \ \underset{y}{\mathbf{F}}\ \ \underbrace{\mathbf{I\ B\ I\ B\ I\ B}}_{\text{3 elements of z}}$$

4.4 Object references(pointers)

References have no meaning in the receiver’s memory space. So the entire data structure pointed at must be encoded and transmitted.
Structural information must be maintained and encoded in a linear message.

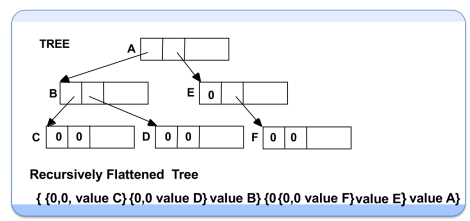
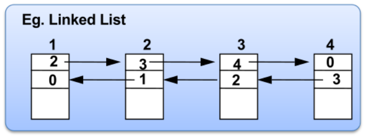


Figure 1: Reference within the object

Structural information must be flattened:
- But must not copy data multiple times.
- Number sub-objects
- Transform pointers into *handles* (ie. number) of sub-objects.

Sub object 1
  Handle 2
  Null
  Contents
Sub object 2
  Handle 3
  Handle 1
  Contents
Sub object 3
  Handle 4
  Handle 2
  Contents



Sub object 4
  Null
  Handle 3
  Contents

12

Figure 2: Reference to objects already transmitted

4.5 Extensible Markup Language (XML)

- **element**: container for data, enclosed by start and end tag. can contain other elements.
- **attribute**: used to label data — usually name/value
- **namespace**: used to scope names
  - defining a set of names each for a collection of element types and attributes referenced by a url
  - specify namespace by `xmlns` attributes
  - can use namespace name as prefix for names
- **schema**: defines elements and attributes that can appear in a document

4.6 JSON

- structural tokens: [{] } : ,
- literal name tokens: `true`, `false`, `null`
- value: `object`, `array`, `number`, `string`, `true`, `false`, `null`.

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1934 </pers:year>
</person>
```

Namespace attribute

Namespace prefix

Figure 3: an example for XML

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions  >
  <xsd:element name= "person" type ="personType" />
    <xsd:complexType name="personType">
      <xsd:sequence>
        <xsd:element name = "name"  type="xs:string"/>
        <xsd:element name = "place"  type="xs:string"/>
        <xsd:element name = "year"  type="xs:positiveInteger"/>
      </xsd:sequence>
      <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

Figure 4: an example for XML Schema

- object: { `string` : `value`, `string` : `value`, ... }
- array: [`value`, `value`, ...]
- string: "⟨sequence of Unicode character⟩" with usual escapes

4.7 Java Object Serialization

- Type (class) information is included with the serialization.
- Reference to other objects are treated as *handles*.
- **Reflection**: the ability to query a class for the name and types of its attributes and methods
- Reflection allows for generic code for marshalling and unmarshalling.

```
{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": 100
    },
    "Animated" : false,
    "IDs": [116, 943, 234, 38793]
  }
}
```

Figure 5: an example for JSON

- steps of serialization
  1. write class information
  2. write types and names of instance variables
  3. if instance variables are of a new class, then repeat the above two steps for those variables
  4. uses serialization

4.8 Liminations

- Representations can have similar syntax but different meaning
  - e.g. rectangular or polar coordinates, transformation is application dependent
- Type may have no meaning outside own context
  - e..g pointer, file name
- Procedures passed as parameters
  - cannot always transfer code to different computer for execution

```java
ByteArrayOutputStream byteobj =
        new ByteArrayOutputStream();
ObjectOutputStream out =
        new ObjectOutputStream(byteobj);
out.writeObject(new Person(“Joe”, “Paris”, 2003);
out.close();
byteobj.close();

byte[] buffer = byteobj.toByteArray();

ByteArrayInputStream inputobj = new ByteArrayInputStream(buffer);
ObjectInputStream in = new ObjectInputStream(inputobj);
Person p = (Person) in.readObject();
in.close();
inputobj.close();
```

Figure 6: an example for Java Object Serialization

5 Remote Procedure Calls (RPCs)

5.1 RPC Interactions

1. Client is suspended until the call completes.
2. Request must include name of the operation and parameters passed *by value*.
3. Server operates locally and sends the result (in one or multiple messages).
4. Client decodes result and returns it to the calling procedure, which then continues processing.

5.2 Stub Procedures

- Client-side definition: the implementation of encoding parameters, send request messages, wait for reply and decode the reply and return.
- Server-side definition: the implementation of receive request, identify local procedure, decode parameters, call the procedure, encode and send the result.

- what **stubs** do in general:
  - parameter marshalling (packing)
  - unmarshal (unpack) received messages and assign values to parameters
  - transform data representations if necessary
  - access communication primitives to send/receive messages.
- Stubs can be automatically generated from an *interface specification*.

5.3 Dispatcher

It maps incoming calls onto relevant procedure (stub). Dispatcher at server receives all "call" messages and uses procedure number (name) to identify called procedure.

5.4 Interface Compiler

- generates a number for each procedure in interface – inserted into call message by client stub procedure.
- generates the stub code and the skeleton code which can then be compiled with the client and the server.
- needs to be specified by the **Interface Definition Language (IDL)** to
  - define the types that can be used
  - define the interfaces and procedures that can be called
  - define the direction of the parameters: in, out, inout
  - mappings to specific languages
  - e.g. the following

```
interface calc {
        void mult ( [in]  float a, [in]  float b, [out] float Res );
        void square ( [in]  float a, [out] float Res );
}
```

5.5 Interface Type Checking

5.5.1 Same Interface

- Identity can be specified, e.g.
  - checksum over source
  - name + timestamp of last modification or compilation
- Client and server hold identity of interface.
- While connecting, check type identities are equal.
- This provides **strong type compatibility**.

5.5.2 Allow subtyping

Permit server to be subtype of client interface, i.e. provides additional operations which are not used by client, but must snot modify operations in original interface.

5.5.3 Structural Compability

Maintain run-time representation of interface and check for structural compatibility when client connects to server. The two interfaces shown in Figure 7 are structurally equivalent. This provides **weak type compatibility**.

```
interface A {
    opa1 (in string a1,
        in short a2 , out long a4);
    opa2 (in string a4);
}

interface B {
    opb1 (in string b1,
        in short b2 , out long b3);
    opb2 (in string b4)
}
```

Figure 7: weak type compatibility example

5.6 Binding

- definition:
  - connecting to a specific server

- assignment of a reference value (e.g. address or object reference) to a placeholder (e.g. message port or object reference variable)

- **Name server** (or **directory server**) is used to register exported interfaces and is queried to locate a server when an interface is imported.
  - When a server starts it *exports* a reference to the interface to the name server.
  - When a client wants to use a service it connects to the name server and *imports* a reference to the server.
- Please refer to the coursework code to see how Java RMI binds client with server.

5.7 Failure

5.7.1 Best Effort (Maybe) semantics

As shown in the following, there is *no fault tolerance measures*.

```
bool call (request, reply) {
   send(request);
   return receive(reply,T)    // return false if timeout;
}
```

The semantics is lightweight, but leaves issues of state consistency of the server, with repsect to the client, up to the application programmer.

5.7.2 At least once semantics

As shown in the following, retries up to n times — if the call succeeds then procedure has been executed once or more times since duplicate messages may have been generated.

```
bool call (request, reply) {
  int retries = n;
  while(retries--) {
        send(request);
        if (receive(reply,T)) return true;   }
  return false; // return false if timeout
}
```

This is useful for **idempotent** server operations, i.e. multiple executions leave the same effect on server state as a single execution.

5.7.3 At most once semantics

- Guarantees that the remote procedure is either not executed or executed once.
- The server must
  - keep track of request identifiers and discard retransmitted requests that have not completed execution.
  - buffer replies and retransmit until acknowledged by the client.
  - not crash to guarantee at-most-once semantics
- It effectively achieves exactly-once semantics if no errors or exceptions have occurred.

Retransmit request	Duplicate Filtering	Re-execute procedure or retransmit reply	Call semantics
No	N/A	N/A	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

Figure 8: Failure semantics comparison

5.7.4 Zero or once (Transactional) semantics

- Guarantees that either the procedure is completely executed or it is not executed at all.
- The server must implement an *atomic transaction for each RPC.*
  - either the state data in the server is updated permanently by an operation taking it from one consistent state to another
  - or it is left in its original state, if the call is aborted or a failure occurs.

- This requires ACID (Atomicity, Consistency, Isolation, Durability) properties and implemented by a *two-phase commit* type of protocol.
  - phase 1: prepares all the aspects of the transaction
  - phase 2: permanently commit or abort them

5.7.5 Server Failure

- Client needs to know server epochs to know if there is server failure leading to loss of state information in the server.
- Use `exportid` to detect failed server: when server restarts a new `exportid` is generated and exported.
- Client receives `exportid` during binding and will include it in all messages to the server.
- Dispatcher aborts calls with incorrect `exportid`.

5.7.6 Client Failure

- **Orphan executions**: result from a client crashing while the server is executing the procedure.
- Server's response will then not be acknowledged. Server either implements a form of rollback or does nothing.
- For long running procedures, to avoid wasting resources, the server may wish to be informed of client crashes so that it can abort orphan executions.

5.8 Implementation

- **TID**: a *transaction identifier* for each invocation. This includes the export identifier.
- **sn**: a message *sequence number* to detect duplicate messages and messages which follow in the sequence of invocations.
- **flag**:
  - `ack`: please acknowledge message
  - `no ack`: no acknowledgement expected
- **params**: in or out parameters as needed.
- Please see Figure 9 for an example of RPC implementation.

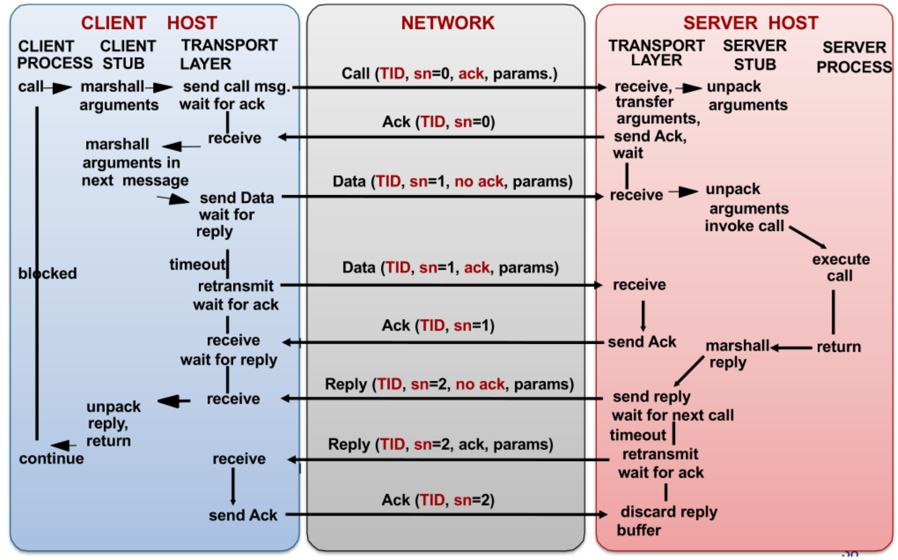


Figure 9: An example of RPC implementation

5.9 Concurrency

- client: no deadlocks with *callbacks* if client multi-threaded.
- server:
  - thread-per-request: dispatcher creates new thread to handle each request
  - thread pool: fixed number of threads generated at start-up, free threads are allocated to requests by the dispatcher. lower creation overhead.
  - thread-per-session: a thread is created at connection set up to process all requests from the particular client.

6 Distributed Object Systems