4.3. External data representation and marshalling

- At language-level data are stored in data structures
- At TCP/UDP-level data are communicated as ‘messages’ or streams of bytes – hence, conversion/flattening is needed
  - Converted to a sequence of bytes
- Problem? Different machines have different primitive data reps,
  - Integers: big-endian and little-endian order
  - float-type: representation differs between architectures
  - char codes: ASCII, Unicode
- Either both machines agree on a format type (included in parameter list) or an *intermediate* external standard is used:
  - External data representation: an agreed standard for the representation of data structures and primitive values
  - e.g., CORBA Common Data Rep (CDR) for many languages; Java object serialization for Java code only

4.3. External data representation and marshalling

- Marshalling: process of taking a collection of data items and assembling them into a form suitable for transmission
- Unmarshalling: disassembling (restoring) to original on arrival
- Three alter. approaches to external data representation and marshelling:
  - CORBA's common data representation (CDR)
  - Java's object serialization
  - XML (Extensible Markup Language) : defines a textual format for rep. structured data
- First two: marshalling & unmarshalling carried out by middleware layer
  - XML: software available
- First two: primitive data types are marshalled into a binary form
  - XML: represented texually
- Whether the marshalled data include info concerning type of its contents?
  - CDR: no, just the values of the objects transmitted
  - Java: yes, type info in the serialized form
  - XML: yes, type info refer to externally defined sets of names (with types), *namespaces*

4.3. External data representation and marshalling

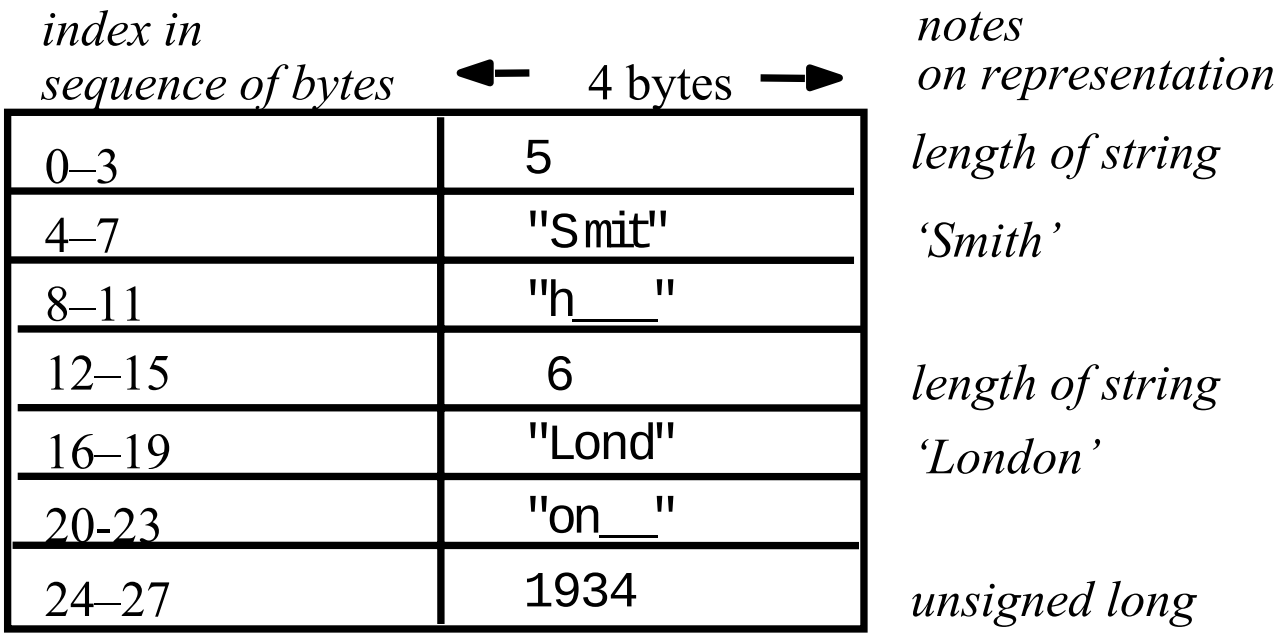
- Although we are interested in the use of external data representation for the arguments and results of RMIs and RPCs, it has a more general use for representing data structures, objects, or structured documents in a form suitable for transmission or storing in files

4.3. External data representation and marshalling

- CORBA CDR
  - 15 primitive types: short, long, unsigned short, unsigned long, float, double, char, boolean, octet, any
  - Constructed types: sequence, string, array, struct, enum and union
    - note that it does not deal with objects (*only Java does: objects and tree of objects*)

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

4.3. External data representation and marshalling



The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

4.3. External data representation and marshalling

- Type of a data item not given: assumed sender and recipient have common knowledge of the order and types of data items
- Types of data structures and types of basic data items are described in CORBA IDL
  - Provides a notation for describing the types of arguments and results of RMI methods

```
Struct Person {
    string name;
    string place;
    unsigned long year;
};
```

4.3. External data representation and marshalling

- Java object serialization
  - Both objects and primitive data values may be passed as arguments and results of method invocations
  - The following Java class is equivalent to Person struct

```
public class Person implements Serializable {
        private String name;
        private String place;
        private int year;
        public Person(String aName, String aPlace, int aYear) {
                name = aName;
                place = aPlace;
                year = aYear;
        }
        // followed by methods for accessing the instance variables
}
```

- Serializable interface (provided in java.io package) allows its instances to be serialized

4.3. External data representation and marshalling

- Serialization: flattening objects into a serial form for storing on disk or transmitting in a message
- Deserialization: restoring the state of objects from serialized form
  - Assumed has no prior knowledge of the types of the objects in the serialized form
  - Some information about the class of each object is included in the serialized form

4.3. External data representation and marshalling

- Java objects can contain references to other objects
  - All objects it references are serialized
  - References are serialized as handles
    - A handle is a reference to an object within the serialized form
    - Each object is written once only
    - Handle is written in subsequent occurrences

4.3. External data representation and marshalling

- To serialize an object:
  - (1) its class info is written out: name, version number
  - (2) types and names of instance variables
    - If an instance variable belong to a new class, then new class info must be written out, recursively
    - Each class is given a handle
  - (3) values of instance variables

  - Example: *Person p = new Person("Smith", "London", 1934);*

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name	java.lang.String place	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

4.3. External data representation and marshalling

- To make use of Java serialization:
  - To serialize: create an instance of ObjectOutputStream
  - Invoke writeObject method passing Person object as argument

  - To deserialize: create an instance of ObjectInputStream
  - Invoke readObject method to reconstruct the original object

```
ObjectOutputStream out = new ObjectOutputStream(… );
        out.writeObject(originalPerson);

ObjectInputStream in = new ObjectInputStream(…);
        Person thePerson = in.readObject();
```

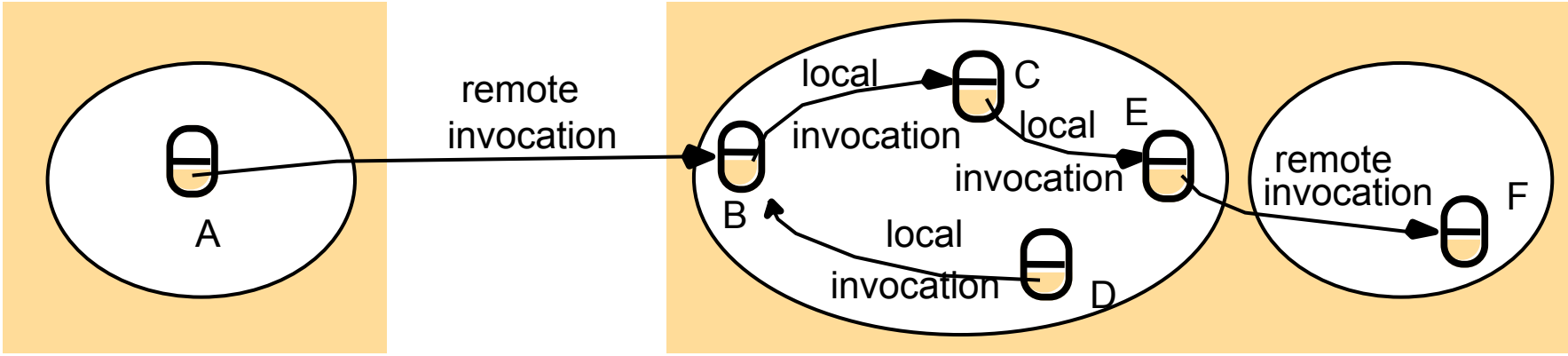
4.3. External data representation and marshalling

- Use of reflection
  - Reflection: inquiring about class properties, e.g., names, types of methods and variables, of objects
  - Allows to do serialization and deserialization in a generic manner, unlike in CORBA, which needs IDL specifications

- For serialization, use reflection to find out (1) class name of the object to be serialized and (2) the names, types and (3) values of its instance variables
- For deserialization, (1) class name in the serialized form is used to create a class, (2) it is then used to create a constructor with arguments types corresponding to those specified in the serialized form. (3) the new constructor is used to create a new object with instance variables whose values are read from the serialized form

4.3. External data representation and marshalling

- Each process contains objects, some of which can receive remote invocations, others only local invocations
- Those that can receive remote invocations are called *remote objects*
  - Java and CORBA support distributed object model
- Objects need to know the *remote object reference* of an object in another process in order to invoke its methods
- The *remote interface* specifies which methods can be invoked remotely
- Remote object references are passed as arguments and compared to ensure uniqueness



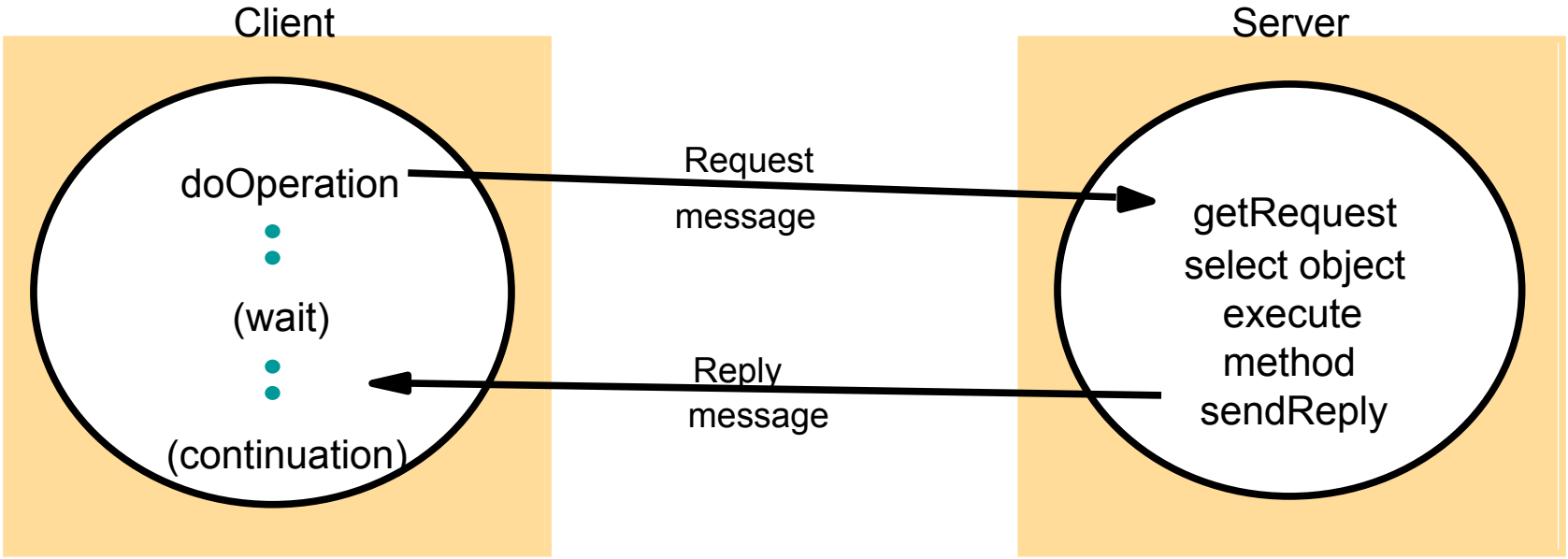
4.3. External data representation and marshalling

- A remote object reference must be unique over space and time
  - Over space: there may be many processes hosting remote objects
  - Over time: It should not be reused after the object is deleted. Why not?
    - its potential invoker may retain obsolete references
- (IP address + port #) + (time of creation + local object number)
  - local object number is incremented each time an object is created in that process
  - identifies the object within the process
  - in case objects live only in the process that created them, the reference can be used as an address of the remote object
  - to allow remote objects to be relocated in a different process on a different computer, the reference cannot be used as address
- Its interface tells the receiver what methods it has (e.g. class *Method*)

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

4.4. Client-Server communication

- Designed to support typical client-server interactions
- Request-reply: usually synchronous (why?)
- Request-reply protocol: built over UDP or TCP (unnece. overheads)
  - ack redundant (why?)
  - connection establishing overhead
  - flow control overhead, redundant for majority of invocations, which pass only small arguments and results



4.4. Client-Server communication

- Request-reply protocol: 3 primitives

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.

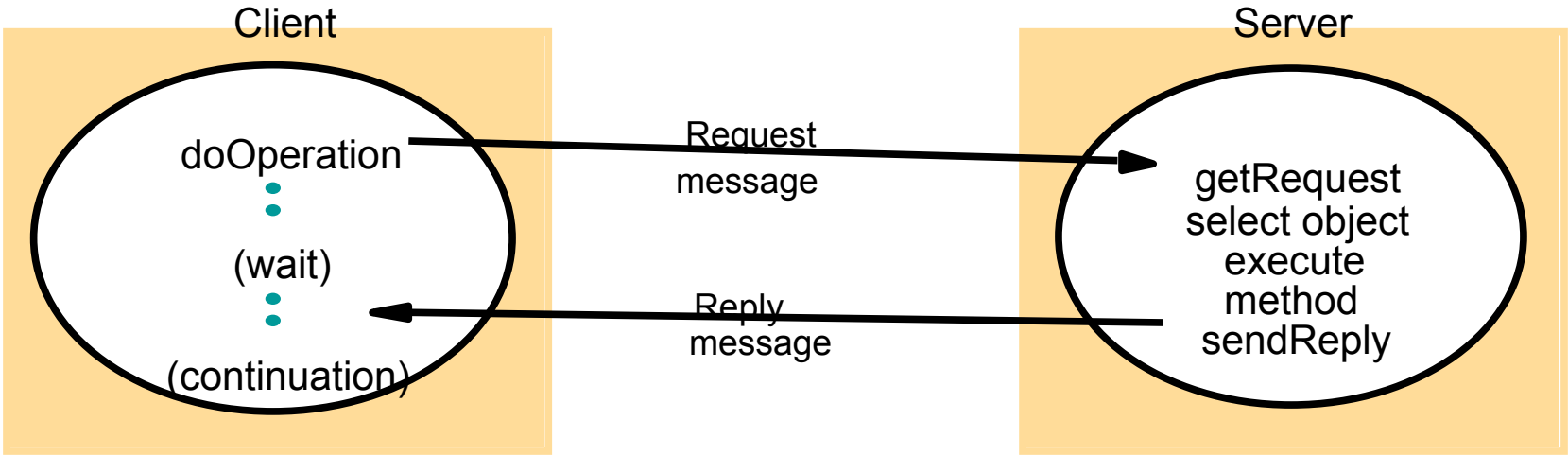
The arguments specify the remote object, the method to be invoked and the arguments of that method.

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.



4.4. Client-Server communication

- Request-reply message structure:

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

- Marshalled RemoteObjectRef:

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

4.4. Client-Server communication

- Message identifiers may be required by some schemes:
  - duplicate request handling
  - requestId: taken from an increasing sequence of integers by the sending process
  - identifier for the sender process: IP address + port #
- Duplicate request handling: (scenario?)
  - if reply not sent: make sure only execute once
  - if reply sent: need to re-execute, two cases
    - a server whose operations are all idempotent, ok
      - idempotent operation: can be performed repeatedly with the same effect as if only performed exactly once (e.g.?)
    - otherwise, use a “history”, record of transmitted messages

Summary

- Heterogeneity is an important challenge to designers:
  - Distributed systems must be constructed from a variety of different networks, operating systems, computer hardware and programming languages
    - The Internet communication protocols mask the difference in networks and middleware can deal with the other differences
- External data representation and marshalling

  - CORBA marshals data for use by recipients that have prior knowledge of the types of its components. It uses an IDL specification of the data types

  - Java serializes data to include information about the types of its contents, allowing the recipient to reconstruct it. It uses reflection to do this
- RMI

  - each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely

  - local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once

  - Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers

4.4. Client-Server communication

- HTTP: an example of a request-reply protocol (TCP based)
- Self-read
- Projects?

- http://www.ida.liu.se/~TDDB37/labs/