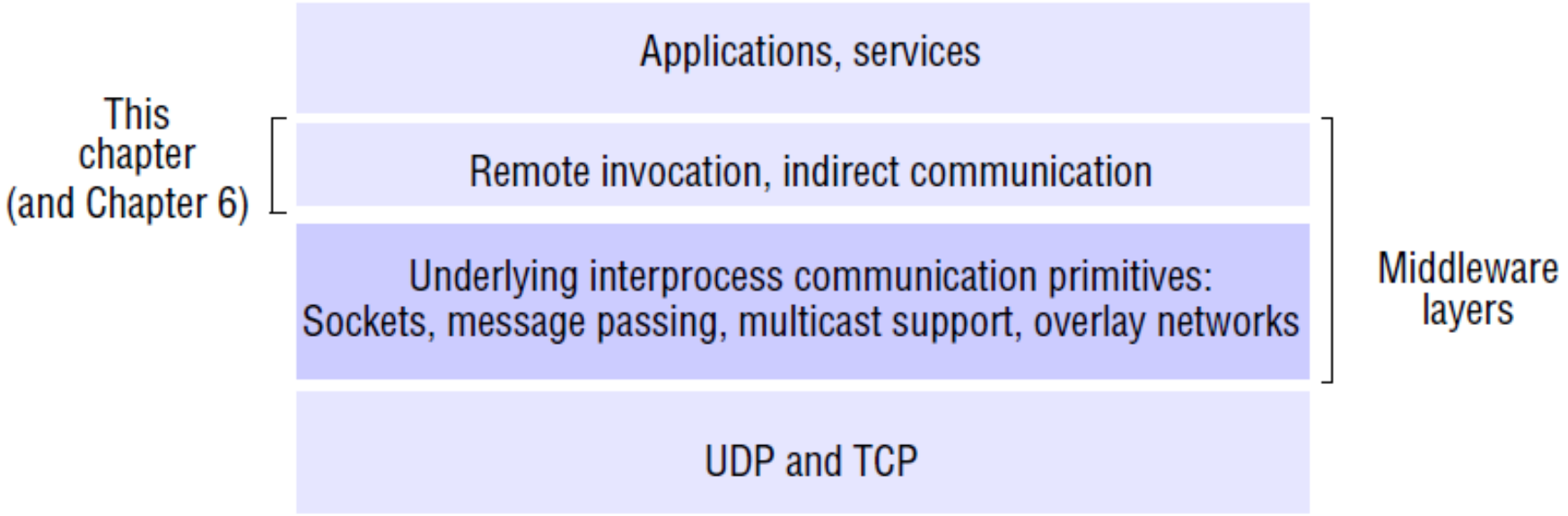
Remote Invocation

The two most prominent remote invocation techniques for communication in distributed systems:

- The remote procedure call (RPC) approach extends the common programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local.
- Remote method invocation (RMI) is similar to RPC but for distributed objects, with added benefits in terms of using object-oriented programming concepts in distributed systems and also extending the concept of an object reference to the global distributed environments, and allowing the use of object references as parameters in remote invocations.



This chapter is concerned with how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system, examining, in particular, the remote invocation paradigms

- ***Request-reply protocols*** represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing.

In particular, such protocols provide relatively low-level support for requesting the execution of a remote operation, and also provide direct support for RPC and RMI, discussed below.

• The earliest and perhaps the best-known example of a more programmer-friendly model was the extension of the conventional procedure call model to distributed systems (the ***remote procedure call***, or RPC, model), which allows client programs to call procedures transparently in server programs running in separate processes and generally in different computers from the client.

• In the 1990s, the object-based programming model was extended to allow objects in different processes to communicate with one another by means of ***remote method invocation*** (RMI). RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.

Request-reply protocols

The client-server exchanges are described in the following paragraphs in terms of the *send* and *receive* operations in the Java API for UDP datagrams, although many current implementations use TCP streams.

A protocol built over datagrams avoids unnecessary overheads associated with the TCP stream protocol. In particular:

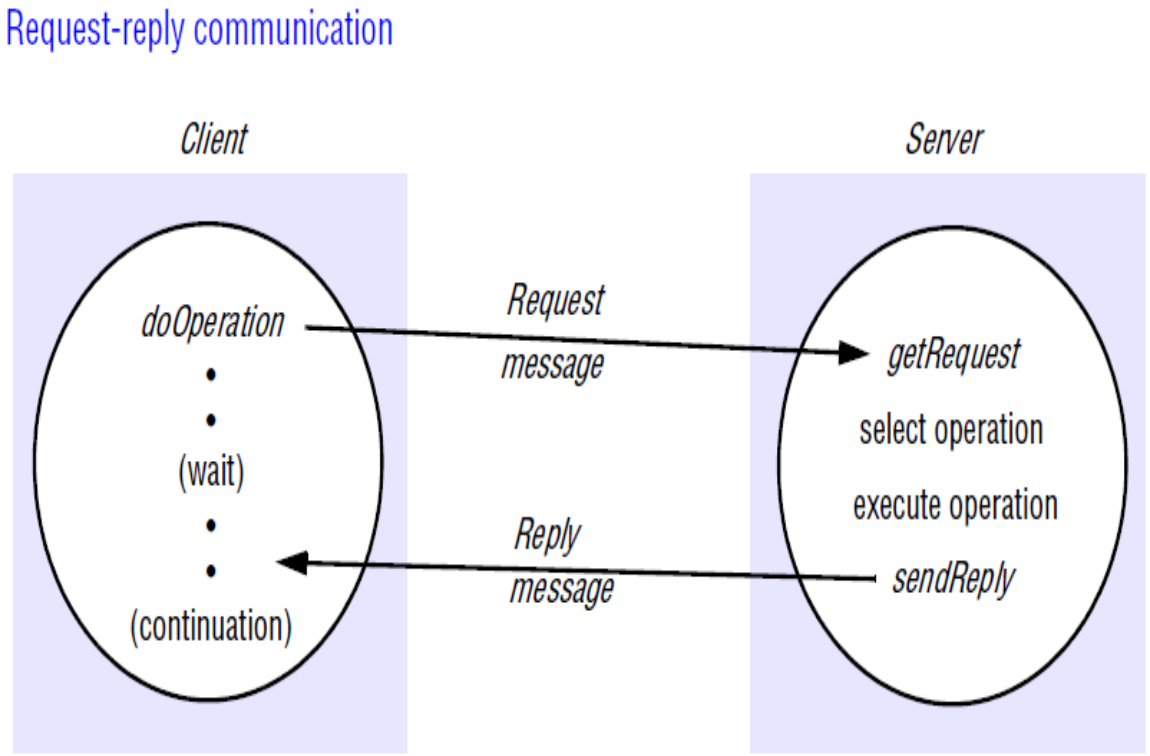
• Acknowledgements are redundant, since requests are followed by replies.

• Establishing a connection involves two extra pairs of messages in addition to the pair required for a request and a reply.

• Flow control is redundant for the majority of invocations, which pass only small arguments and results.

The request-reply protocol • The protocol we describe here is based on a trio of communication primitives, *doOperation*, *getRequest* and *sendReply*, as shown in Figure 5.2 . This request-reply protocol matches requests to replies. It may be designed to provide certain delivery guarantees.

If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message. Figure 5.3 outlines the three communication primitives.



Operations of the request-reply protocol

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.

The *doOperation* method is used by clients to invoke remote operations.
Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation.
Its result is a byte array containing the reply.
It is assumed that the client calling *doOperation* marshals the

arguments into an array of bytes and unmarshals the results from the array of bytes that is returned.

The first argument of *doOperation* is an instance of the class *RemoteRef*, which represents references for remote servers.

This class provides methods for getting the Internet address and port of the associated server.

The *doOperation* method sends a request message to the server whose Internet address and port are specified in the remote reference given as an argument.

After sending the request message, *doOperation* invokes *receive* to get a reply message, from which it extracts the result and returns it to the caller.

The caller of *doOperation* is blocked until the server performs the requested operation and transmits a reply message to the client process.

getRequest is used by a server process to acquire service requests, When the server has invoked the specified operation, it then uses *sendReply* to send the reply message to the client.

When the reply message is received by the client the original *doOperation* is unblocked and execution of the client program continues.

The information to be transmitted in a request message or a reply message is shown in Figure

The first field indicates whether the message is a *Request* or a *Reply* message.

The second field, *requestId,* contains a message identifier. A *doOperation* in the client generates a *requestId* for each request message, and the server copies these IDs into the corresponding reply messages.

This enables *doOperation* to check that a reply message is the result of the current request, not a delayed earlier call.

The third field is a remote reference.

The fourth field is an identifier for the operation to be invoked.

For example, the operations in an interface might be numbered 1, 2, 3, ... , if the client and server use a common language that supports reflection, a representation of the operation itself may be put in this field

Request-reply message structure

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

•Reflection is an API that is used to examine or modify the behavior of methods, classes, and interfaces at runtime. The required classes for reflection are provided under **java.lang.reflect** package. Reflection gives us information about the class to which an object belongs and also the methods of that class that can be executed by using the object.

•Through reflection, we can invoke methods at runtime irrespective of the access specifier used with them.

Message identifiers • Any scheme that involves the management of messages to provide additional properties such as reliable message delivery or request-reply communication requires that each message have a unique message identifier by which it may be referenced.

A message identifier consists of two parts:
1. a *requestId*, which is taken from an increasing sequence of integers by the sending process;
2. an identifier for the sender process, for example, its port and Internet address.

The first part makes the identifier unique to the sender, and the second part makes it unique in the distributed system. (The second part can be obtained independently – for example, if UDP is in use, from the message received.)
When the value of the *requestId* reaches the maximum value for an unsigned integer (for example, $2^{32} - 1$) it is reset to zero.

Failure model of the request-reply protocol • If the three primitives *doOperation*, *getRequest* and *sendReply* are implemented over UDP datagrams, then they suffer from the same communication failures.

That is:

• They suffer from omission failures.

• Messages are not guaranteed to be delivered in sender order.

Omission failure happens when a crucial requirement or feature is accidentally or intentionally omitted from the design.

In addition, the protocol can suffer from the failure of processes

. We assume that processes have crash failures. That is, when they halt, they remain halted – they do not produce Byzantine behaviour

A Byzantine fault is **a state of a computer system, particularly distributed computing systems, where components may fail and there is imperfect information on whether a component has failed**.

Timeouts • There are various options as to what *doOperation* can do after a timeout.
The simplest option is to return immediately from *doOperation* with an indication to the client that the *doOperation* has failed.
This is not the usual approach – the timeout may have been due to the request or reply message getting lost and in the latter case, the operation will have been performed.
To compensate for the possibility of lost messages, *doOperation* sends the request message repeatedly until either it gets a reply or it is reasonably sure that the delay is due to lack of response from the server rather than to lost messages.
Discarding duplicate request messages • In cases when the request message is retransmitted, the server may receive it more than once.
For example, the server may receive the first request message but take longer than the client's timeout to execute the command and return the reply.
This can lead to the server executing an operation more than once for the same request. To avoid this, the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter out duplicates.
If the server has not yet sent the reply, it need take no special action – it will transmit the reply when it has finished executing the operation.

Lost reply messages • If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result, unless it has stored the result of the original execution.

Some servers can execute their operations more than once and obtain the same results each time. An *idempotent operation* is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once.

A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once.

History • For servers that require retransmission of replies without re-execution of operations, a history may be used. The term 'history' is used to refer to a structure that contains a record of (reply) messages that have been transmitted.

As clients can make only one request at a time, the server can interpret each request as an acknowledgement of its previous reply.

Therefore the history need contain only the last reply message sent to each client. However, the volume of reply messages in a server's history may still be a problem when it has a large number of clients.

This is compounded by the fact that, when a client process terminates, it does not acknowledge the last reply it has received – messages in the history are therefore normally discarded after a limited period of time

RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

In the R protocol, a single *Request* message is sent by the client to the server.
The R protocol may be used when there is no value to be returned from the remote operation and the client requires no confirmation that the operation has been executed.
The client may proceed immediately after the request message is sent as there is no need to wait for a reply message.
This protocol is implemented over UDP datagrams and therefore suffers from the same communication failures.

The RR protocol is useful for most client-server exchanges because it is based on the request-reply protocol.
Special acknowledgement messages are not required, because a server's reply message is regarded as an acknowledgement of the client's request message.
Similarly, a subsequent call from a client may be regarded as an acknowledgement of a server's reply message
The RRA protocol is based on the exchange of three messages: request-replyacknowledge reply.
The *Acknowledge reply* message contains the *requestId* from the reply message being acknowledged.
This will enable the server to discard entries from its history. The arrival of a *requestId* in an acknowledgement message will be interpreted as acknowledging the receipt of all reply messages with lower *requestId*s, so the loss of an acknowledgement message is harmless