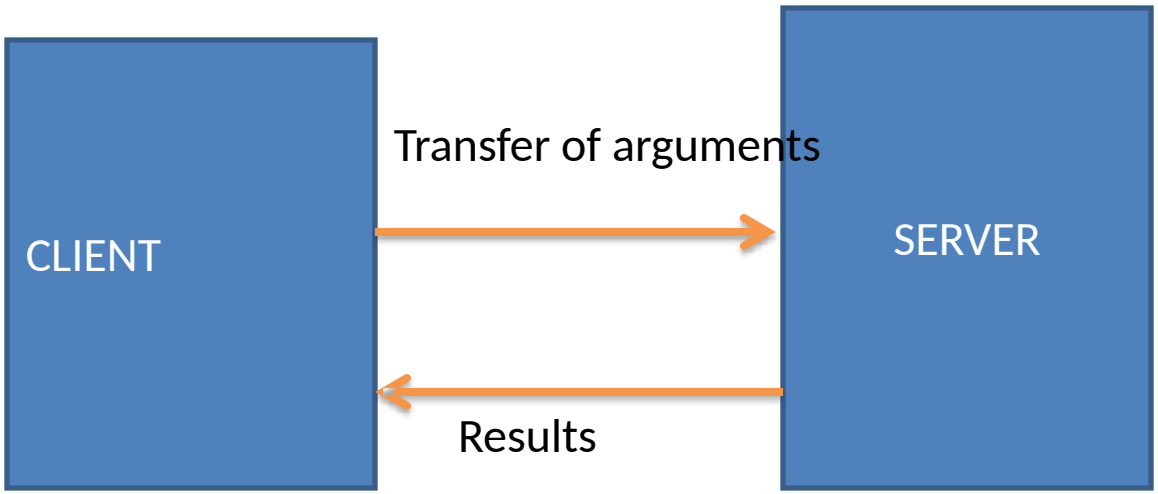
Marshalling Arguments

- Implementation of remote procedure calls



Marshalling Arguments

- Transfer of message data requires encoding and decoding of the message data.
- For RPCs this operation is known as *Marshaling* and involves the following Actions:

1. Taking the arguments of a client process or the result of a server

1. Encoding the message data of step 1 above on the sender's computer.This encoding process involves the conversion of program objects into a stream form that is suitable for transmission and Placing them into a message buffer.

2. Decoding of the message data on the receiver's Computer.The reconstruction of program objects from the message data that was received in stream form.

Marshalling Arguments

- Marshalling procedure may be classified as
  1. Provided as a part of the RPC software- Marshalling procedures for scalar data types and compound types build from the scalar ones
  2. Those that are defined by the users of the RPC system- Marshalling procedures for user defined data types and data types that include pointers
- A good RPC system
  - generate in-line marshaling code for every remote call
  - it is difficult to achieve this goal because of the large amounts of code

SERVER MANAGEMENT

- Issues in server management are server implementation and server creation.
- **Server Implementation**

1. Stateful Servers –Client state info is stored

  – Open *(filename, mode):* This operation is used to open a file identified by *filename* in the specified *mode.*

  – Read (f*id, n, buffer):* This operation is used to get *n* bytes of data from the file

  – Write *(fid, n, buffer):* On execution of this operation, the server takes *n* bytes of data

  – Seek *(fid, position):* causes the server to change the value of the *read write pointer*

  – Close *(fid):* This statement causes the server to delete from its *file-table* the file state

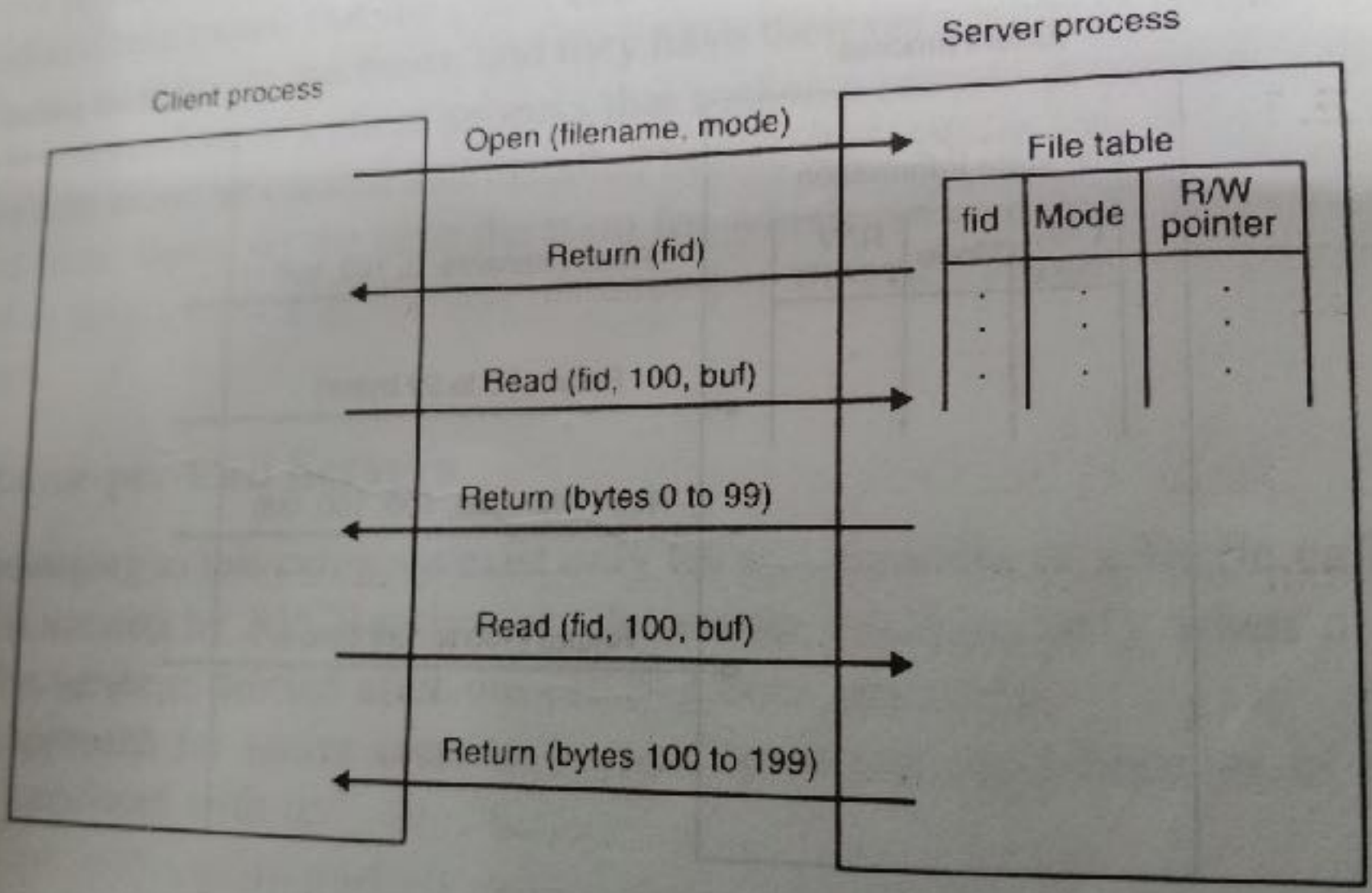


Fig. 4.5 An example of a stateful file server.

Stateless server

Read (filename,position,n,buffer)

Write(filename,position,n,buffer)

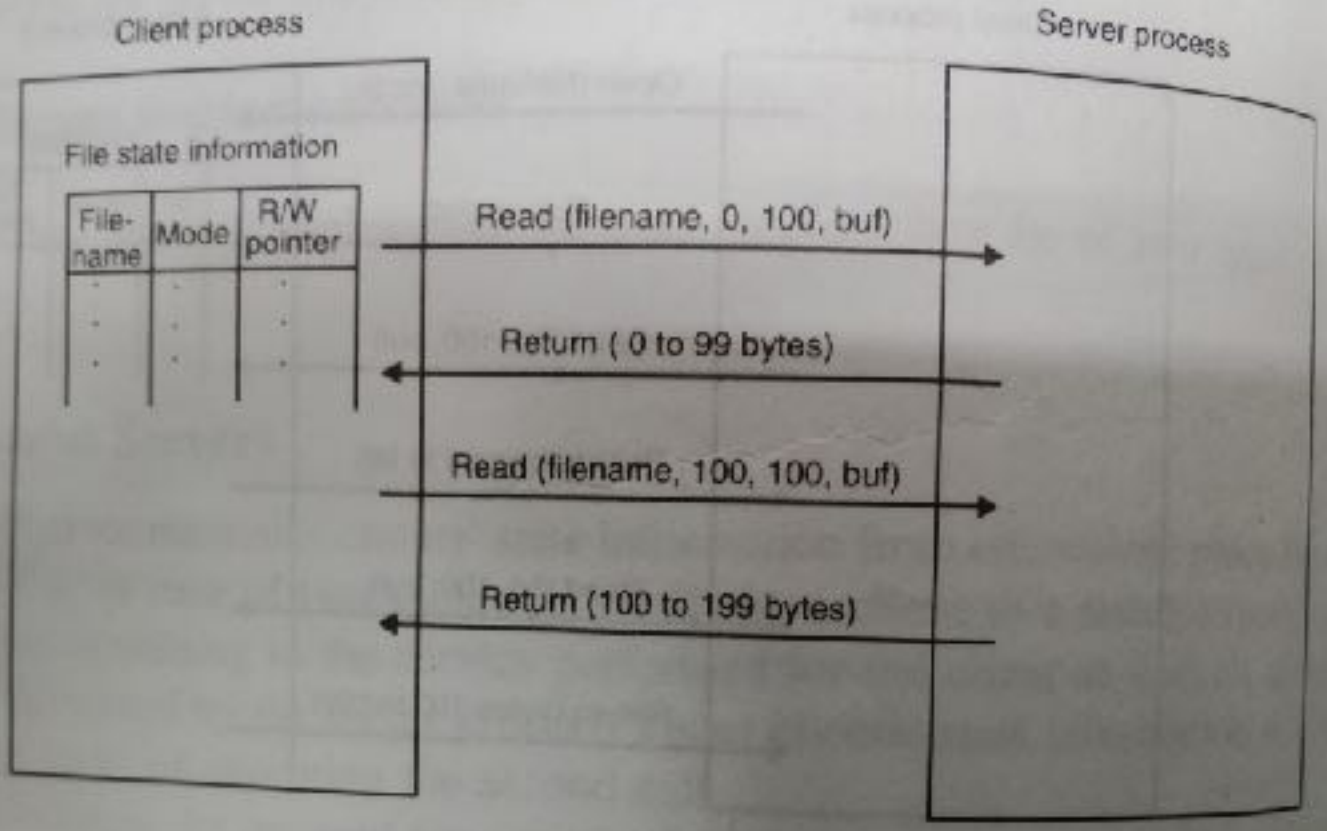


Fig. 4.6 An example of a stateless file server.

Server Creation Semantics

- The remote procedure to be executed is totally independent of the client process.

Based on the time duration for which RPC servers survive, they may be classified as

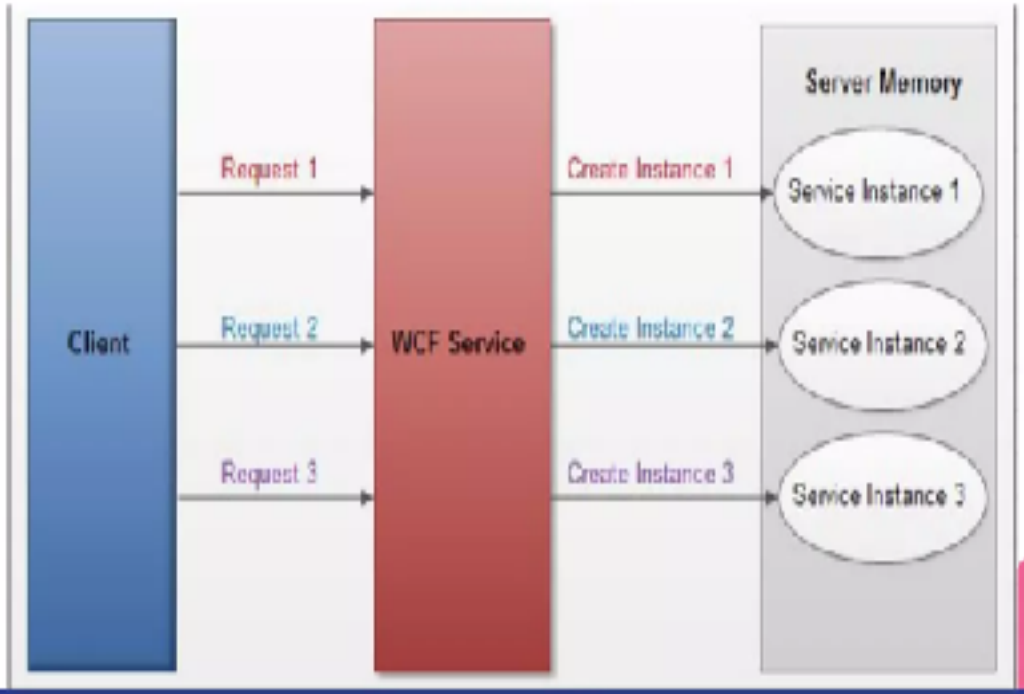
- Instance-per-call servers,
- Instance-per-transaction or Session servers
- Persistent servers.

Server Creation Semantics

- Instance-per-Call Servers
  - Servers belonging to this category exist only for the duration of a single call.
  - A server of this type is created by RPC Runtime on the server machine only when a call message arrives.
  - The server is deleted after the call has been executed.

1.Instance-per-Call Servers

- This Category of Servers exists only for the duration of single Call.
- A server of this type is created by RPC Runtime on the server machine only when a call message arrives and deleted after the call has been executed.



Server Creation Semantics

- The servers of this type are stateless because they are killed as soon as they have serviced.
- The involvement of OS to preserve inter call state information will make the remote procedure calls expensive (resource allocation and de alloaction done multiple times)
- If it is maintained by the client process, the state information must be passed to and from the server with each call.
- Will lead to the loss of data abstraction across the client-server
- When same server has to be invoked successively – more expensive.

Server Creation Semantics

- Instance-per-Session Servers
  - Servers exist for the entire session
  - can maintain inter-call state information
  - The overhead involved in server creation, destruction for a client-server session that involves a large number of calls is also minimized.
  - There is a **server manager** for each type of service. Server managers are registered with **binding agent**. Client contacts binding agent **which in turn provides details of server manager** based on the service request
  - **Client contacts server manager** asks to create server for it. After creation passes back its address to the client. Client contacts server directly and **destroyed by server** manager when client informs that no longer needed

Server Creation Semantics

- Persistent Servers
  - Can be shared by more than one clients – No creation only sharing takes place
  - Servers are usually created and installed before the clients that use them.
  - **Client contact binding agent**-Minimum number of clients currently bound to it and returns the address of the selected server to the client.
  - The client then directly interacts with that server.
  - Manage several sets of state information.
  - Improves performance and reliability

Call SEMANTICS

Normal functioning of an RPC may get disrupted due to

- The call message gets lost.
- The response message gets lost.
- The callee node crashes and is restarted.
- The caller node crashes and is restarted.

Call SEMANTICS

Possibly or may be

- This is the weakest semantics
- In *this* method, to prevent the caller from waiting indefinitely for a response from the callee, a timeout mechanism is used.
- The caller waits until a pre-determined timeout period and then continues with its execution.
- Does not guarantee anything about the receipt of the call message.
- The response message is not important for the caller

Call SEMANTICS

- **Last one – Retransmission**
  - suppose process PI of node *N1* calls
  - procedure *Fl* on node *N2*, which in turn calls
  - procedure *F2* on node *N3.(started executing)*
- Node *N1* crashes.
  - Node *N1* 's processes will be restarted, and
  - PI's call to *F1* will be repeated. The second invocation of FI will again call procedure *F2* on node *N3*. Unfortunately, node *N3* is totally unaware of node *N1* crash.
  - Therefore procedure *F2* will be executed twice on node *N3* and *N3* may return the results of the two executions of *F2* in any order

Call SEMANTICS

- The basic difficulty in achieving last-one semantics is caused by orphan calls.
- An *orphan call* is **one whose parent (caller) has expired due to a node crash**.
- To achieve last-one semantics, these orphan calls must be terminated before restarting the crashed processes
- Killing by “orphan extermination”

Call SEMANTICS

- **Last of many**
  - A simple way to neglect orphan calls is to use call identifiers to uniquely identify each call. When a call is repeated, it is assigned a new call identifier.
  - Each response message has the corresponding call identifier associated with it.
  - A caller accepts a response only if the call identifier associated with it matches with the identifier of the

  Most recently repeated call; otherwise it ignores the response message.

Call SEMANTICS

- **Atleast once**
  - This is an even weaker call semantics than the last-of-many call semantics.
  - Guarantees that the call is executed one or more times but does not specify which results are returned to the caller.
  - can be implemented simply by using timeout- based retransmissions
  - if there are any orphan calls, it takes the result of the first response message and ignores the others, whether or not the accepted response is from an orphan.

Call SEMANTICS

- **Exactly once**
  - This is the strongest and the most desirable call semantics because it eliminates the

  Possibility of a procedure being executed more than once
  - No matter how many times a call is retransmitted. The last-one, last-of-many, and at-least-once call semantics cannot guarantee this

Call SEMANTICS

- The main disadvantage of these cheap semantics is that, if a procedure is executed more than once with the same parameters, the same results and side effects will be produced
  - ReadNextRecord(Filename)
  - ReadRecordN(Filename, N)
  - AppendRecord(Filename, Record)
  - GetLastRecordNo(Filename)

  WriteRecordN(Filename, Record, N)

  To append

  Last=GetLastRecordNo(Filename)

  WriteRecordN(Filename, Record, last)