COMMUNICATION AND INVOCATION:-

Communication is not an end in it-self, but is normally part of the implementation. Such as a remote procedure call, whose purpose is to bring about the processing of data in a different scope of execution environment?

Application imposes a variety of demands upon a communication system. These include procedure-consumer, client-server and group communication. They vary as to the quality of service required, that is delivery guarantees, bandwidth the latency, and security provided by the communication service.

Communication primitives:-

Distributed operating system kernels normally provide message passing the *send-receive* combination and the *Do Operation-Get Request-Send Reply* combination. In some systems, however, *Send* is reliable. Where a system provides only one of the two models, it is on the grounds that it can be used to implement the other. For example, in Amoeba the asynchronous semantics of send can be reproduced by

(a) copying the contents of the given message into a dynamically allocated message buffer, so that the caller of send can re-use its buffer; and

(b) Using an independent thread which executes DoOperation while the first thread proceeds.

In the case of *Send*, no thread manipulation is required to achieve asynchronous semantics.

In addition, group communication is provided in several distributed operating systems, including Amoeba, the V system and chorus. The V system provides a multicast equivalent of *DoOperation*, which receives just one reply by default, even though each recipient can reply. Any further replies can be received by a separate call made by the client.

Memory sharing:-

Mach applies copy-on-write memory sharing to transfer of large message between local processes. A message may be constructed from an address space region, which consists of set of entire pages. When the message is passed to local process, a region is created in its address space to hold the message, and this region is copied from the sent region in copy-on-write mode.

Shared regions may be used for rapid communication between a user process and the kernel, or between user processes. Data are communicated by writing to and reading from the shared region. Data are thus passed efficiently; with out copying them to and from the kernels address space.

Protocols and Openness:-

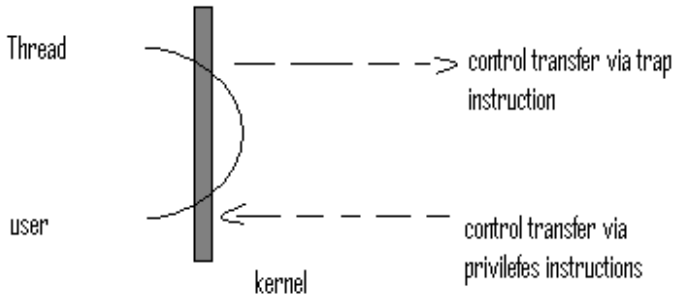
Kernels differ in whether or not they support network communication directly. Some, notably Amoeba, the V system and Sprite, incorporate their own network protocols tuned to RPC interactions-Amoeba RPC, VTMP and Sprite RPC, respectively

Protocols such as TCP, UDP and IP, on the other hand, are widely used over LANs and WANs but do not directly support RPC interactions. Rather than design their own protocols, or commit the kernel to any particular established protocol, the designers of the Match and chorus kernels decided to make the communication design open.

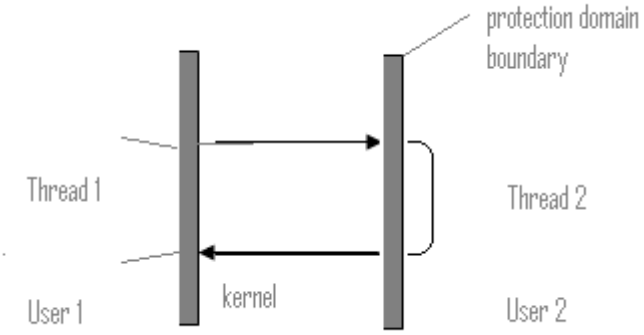
INVOCATION PERFORMANCE:-

Calling a local procedure, making a system call, sending a message, remote procedure calling and invoking a method in an object by sending a message to it, are all examples of invocation mechanisms. Each mechanism causes code to execute to it, are all examples of invocation mechanisms. Each mechanism causes code to be executed out of scope of the calling procedure or object. Each involves, in general, the communication of arguments to this code, and the return of data values to the caller. Invocation mechanisms can be either synchronous, as for example in the case of local or remote procedure calls, or they can be asynchronous, when for example an operation with no return values is invoked upon an object.

(a) System call



(b)RPC (within one computer)



(c)RPC (between computers)

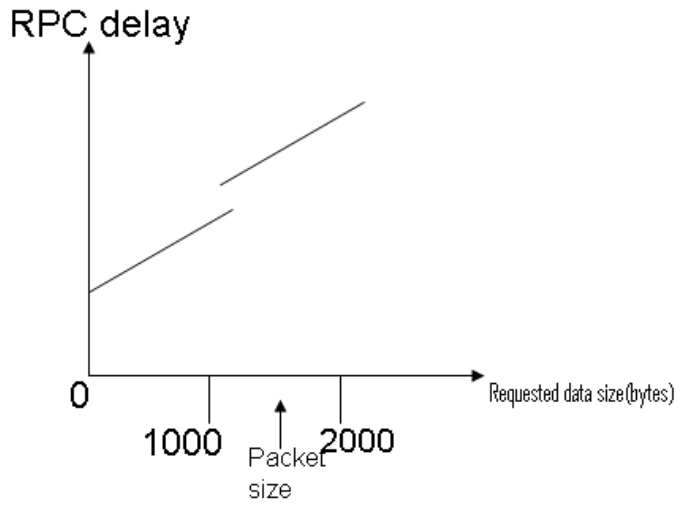
The important performance-related distinctions between invocation mechanisms, apart from whether or not they are synchronous, are: whether they involve communication across a network they involve a domain transition, whether they involve communication across a network, and whether they involve thread scheduling and switching. The above fig shows the particular cases of a system call, an RPC between process hosted by the same computer, and RPC between remote processes.

RPC performance:-

A null RPC is defined as an RPC without parameters that executes a null procedure and returns no values. Its execution involves an exchange of messages carrying little system data and no user data. Currently, the best reported time for a null RPC between two user processes across a LAN is about one millisecond.

Consider an RPC that RPC that fetches a specified amount of data from a server. It has one integer request argument, specifying the size of data required. It has two reply arguments, an integer specifies successes are failure, and when the call is successful an array of bytes from the server.

The below figure shows silent delay against requested data size. The delay is roughly proportional to the size until the size reaches a threshold at about network packet size. Beyond the threshold, at least one extra packet has to be sent, to carry the extra data. Depending on the protocol, a further packet might be used to acknowledge this extra packet. Jumps in the graph occur each time the number of packets increases.



RPC delay against parameter size

The following are the main components accounting for RPC delay, besides network transmissions times:

Marshalling: Marshalling and unmarshalling, which involve copying and converting data, become a significant overhead as the amount of data grows.

Data Copying: The processors in a present-day workstation can data from memory at about 10 megabytes per second. This is about the same as transfer rate that can be achieved on a 100 megabits-per-second network. Potentially, even after marshalling, data is copied several times in the course of an RPC:

1. Across the user-kernel boundary, between the client or server address space and kernel buffers;
2. Across each protocol layer(for example, RPC/UDP/IP/Ethernet);
3. Between the network interface and kernel buffers.

   Transfers between the network interface and main memory are usually handled by direct memory access (DMA). The other copies have to be handled by the processor.

Packet initialization: This involves initializing protocol headers and trailers including checksums. The cost is therefore proportional to the amount of data sent.

Thread scheduling and context switching: **These** may occur as follows:

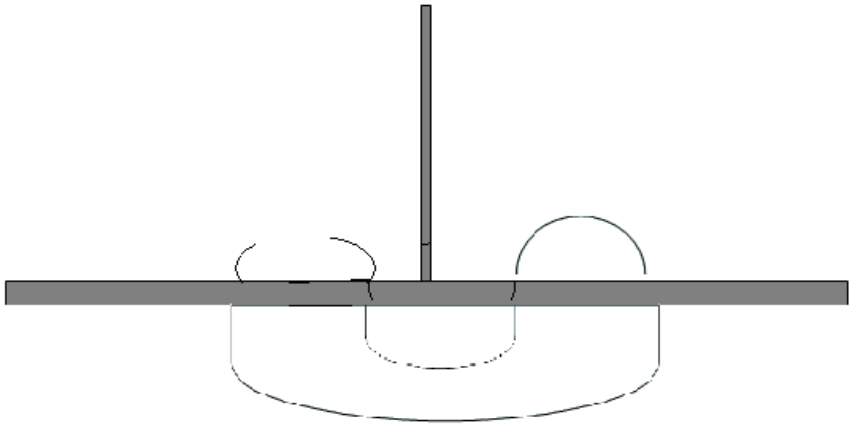
1. Several system calls(that is, context switches)are made during an RPC, as stubs invoke the kernel's communication operations;
2. A server thread is scheduled to call the remote procedure;
3. If the operating system employs a separate network manager process, then each *send* involves a context switch to one of its threads

Waiting for acknowledgements: The choice of RPC protocol may influence delay, particularly when large amounts of data are sent.

RPC within a computer:

Bershad et al. [1990] development a more efficient local invocation mechanism called ***light weight RPC*** (LRPC) based on optimizations concerning data copying and thread scheduling.

First, they noted that it would be more efficient to use shared memory regions for client-server communication, with a different (private) region between the server and each of its local clients. Such a region contains one or more A (for argument) *stacks*. Instead of RPC parameters being copied between the kernel and user address spaces involved, client and server are able pass arguments and return values directly via an A stack. The same stack is used by the client and server stubs. In LRPC, arguments are copied once: when they are marshalled onto the A stack. In an equivalent RPC, they are copied four times: from the client stubs stack onto a message; from the message to a kernel buffer to server message; from the message to the server stub's stack. There may be several A stacks in shared region, became several threads in the same client may call the server at the same time.



An invocation is shown in above figure. A client thread enters the server's execution environment by first trapping to the kernel and presenting it with a capability. The kernel checks this and only allows a context switch to a valid server procedure; if it is valid, the kernel switches the thread's context to call the procedure in the server's execution environment. Entering a body of code from a lower layer (the kernel) in this way is sometimes called an *up call*. When the

procedure in the server returns, the thread returns to the kernel, which switches the thread back to the client execution environment. Note that clients and servers employ stub procedures to hide the details just described from application writers.

Virtual memory:

Virtual memory is the abstraction of single-level storage that is implemented transparently, by a combination of primary memory, such as RAM chips, backing storage, that is a high speed persistent storage mechanism such as a disk.

Much of the implementation of virtual memory in a distributed system is common to that found in a conventional operating system. The main difference is that the backing store interface is to a server, instead of a local disk. General features of the virtual memory concept and implementation that are applicable to distributed systems.

A central aim of virtual memory systems is to be able to execute large programs, and combinations of programs, whose entire code and data are too large to be stored in main memory at any one time. By storing only those sections of code and data currently being accessed by processes, it is possible

1. To run programs whose associated code and data exceeds the capacity of main memory,
2. To increase the level of multiprogramming by increasing the number o process whose working code and data can be stored in main memory simultaneously, and
3. To remove the concerns of physical memory limitations from programmers.

The most common implementation of virtual memory is called *demand paging*. Each page is fetched into primary memory upon primary memory upon demand: that is, when a process attempts to read or write data in a page which is not currently resident, it is fetched from backing store.

A virtual memory system is required to make decisions in two areas. First, its frame allocation policy is an algorithm for deciding how much main memory should be allocated to each running process. Secondly, a page Replacement policy is used when a page must be fetched from secondary storage and there is no room in the main memory cache. A page is chosen to be replaced by the page to be bought in. the virtual memory system applies its policies at two points in the system's operation:

a. When a process attempts to reference a non-resident page, causing a page fault to be raised and handled by the kernel, and
b. Periodically, upon measurement of page fault rates and/or each process's page reference patterns.

External pagers:-

In a distributed system, the computer running a process that incurs a page fault is not necessarily the same computer that manages the corresponding page data. The natural development for virtual memory implementation in distributed systems is for page data to be stored by a server, and not directly by the kernel using a local secondary storage device. These user-level servers are variously called **external pagers** or *external mappers or memory managers.*

Recall that a virtual address space is organized in regions. To consider a general model, we shall assume that any region in an address space is mapped to part or all of some underlying **memory object.** A memory object is a contiguous, addressable resource such as a file or a set of pages managed by an external pager. If the region is an execution stack, for example, then the underlying memory object is one which a file has been mapped, the underlying memory object is one which persists only as long as the process executes.

To map a memory object into a region, the process sends a request to the external pager that manages the memory object. After the mapping, messages pass between the kernel and the external server to deal with paging. The kernel fetches initial data values from the external pager, and pages data to it.

The kernel retains responsibility for handling page fault exceptions generated by local process. It is responsible for main memory management, and therefore for implementing a frame allocation policy. The kernel is normally left to implement its own page replacement policy. The information necessary for applying the page replacement policy, such as bits set by the memory management unit when pages are referenced or modified, is local to the kernel.

The roles of the external pager are:

i. To receive and deal appropriately with data that have been purged by a kernel form its cache of pages, as part of the kernel's page replacement policy;
ii. To supply page data as required by a kernel to complete its page fault handling; and
iii. To impose consistency constraints determined by the underlying memory object abstraction, given that several kernels might attempt to cache modifiable pages of the object simultaneously.

In summary, virtual memory is frame work for accessing any collection of memory objects that can be mapped to individual regions. The common requirement for any memory object abstraction is that is consists of contiguously addressable data items, which may be read and modified. A message passing protocol between the kernel and an external pager

FILE SEVICE

Most application of computer use files for the permanent storage of information or as a means for sharing information between different users and programs. The file is an abstraction of permanent storage. Since the introduction of disk storage in the 1960s, operating systems have included a **file system** component that is responsible for the organization, storage, naming, sharing and protection of files. File systems provide a set of programming operations. File storage is implemented on magnetic disks and other non-volatile storage media.

In most file systems, a file is defined as a sequence of similar-sized data items and the file system provides functions to read and write sub-sequences of data items beginning at any point in the sequence.

File systems are designed to store and mange large numbers of files, with facilities for creating, naming and deleting the files. The naming of files is supported by the use of directories. A **directory** is a file, often of a special type, that provides a mapping from text names to internal identifiers. In most file systems directories may include the names of other directories, leading to the familiar hierarchic file naming scheme and the multi-part *pathnames* for files used in UNIX and other operating systems. File systems also take responsibility for the control of access to files; restricting the access to files according to user's authorizations and the type of access requested.

FILE SYSTEM MODULES

Directory module	Relates file names to file IDs
File module	Relates file IDs to particular files
Access control module	Checks permission for operation requested
File access module	Reads or writes file data or attributes
Block module	Accesses and allocates disk blocks
Device module	Disk I/O and buffering

The above table shows a typical layered module structure for the implementation of a file system as a component of a conventional operating system. Each layer depends only on the layers below it.

Distributed file service requirements:-

A distributed **file service** is an essential component in distributed systems, fulfilling a function similar to the file system component in conventional operating systems. It can be used to support the sharing of persistent storage and of information; it enables user programs to access remote files without copying them to local disk and it provides access to files from disk less nodes. Other services, can be more easily implemented when they can call upon the file service to meet their needs for persistent storage.

The file service is usually the most heavily-used service in a general-purpose distributed system, so it's functionally and performances are critical. The design of the file service should support many of the transparency requirements for distributed systems. The following forms of transparency are partially or wholly addressed by most current file services:

Access transparency: Client program should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

Location transparency: Client programs should see a uniform file name space. Files or groups of files may be relocated with out changing their pathnames, and user programs see the name space wherever they are executed.

Concurrency transparency: Changes to file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control. The need for concurrency control for access to shared data in many applications is widely accepted and techniques are known for its implementation but they are costly. Most current file services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking.

Failure transparency: the correct operation of servers after the failure of a client and the correct operation of client programs in the face of the lost messages and temporary interruptions of the service are the main goals. For UNIX-like file services these can be achieved by the use of stateless servers and repeatable service operations. More sophisticated modes of fault tolerance.

Performance transparency: Client programs should continue should to perform satisfactorily while the load on the service varies within a specified a range.

These are two other important requirements that affect the usefulness of a distributed file service:

Hardware and operating system heterogeneity: The service interface should be defined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of *openness.*

Scalability: the service can be extended by incremental growth to deal with a wide range of loads and network sizes.

The following forms of transparency are also required if scalability is extended to include networks with very large numbers of active nodes. There is as yet no file services that achieve all of them fully, although most recently-developed file services address some of them.

Replication transparency: A file may be represented by several copies of its contents at different locations. This has two benefits-it enables multiple servers to share the load of providing a service to

clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed.

Migration transparency: Neither client programs nor system administration tables in client nodes to be changed when files are moved. This allows file mobility files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

There are some features not found in current file services that will be important for the development of distributed applications in the feature:

Support for fine –grained distribution of data: As the sophistication of distributed application grows, the sharing of data in small units will become necessary. This is a reflection of the need to locate individual objects near the processes that are using them and to cache them individually in those locations. The file abstraction, which was developed as a model for permanent storage in centralized systems doesn't address this need well.

Tolerance to network partitioning and detached operation: Network partitions may be the result of faults, or they may occur deliberately, as for example when a portable workstation is taken away. When a file service includes the replication or caching of files, clients may be affected when a network partition occurs. For example, many replication algorithms require a majority of replicas to respond to request for the most up-to-date copy of a file. If there is a network partition, a majority may not be available, preventing the clients from proceeding.

File service components

The scope for open, configurable systems is enhanced if the file service is structured as three components- **a flat file service**, **a directory service,** and **a client module.** The relevant modules and their relationships. The flat file service and the directory service each export an interface for use by client programs and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files. The client module integrates the flat file service and the directory services. Providing a single programming interface with operations on files similar to those found in conventional file system. The division of responsibilities between the modules can be defined:

Flat file service: The flat file service is concerned with implementing operations on the contents of files. **Unique file identifiers** are used to refer to files in all requests for flat file service is based upon the use of UFIDs. UFIDs are long integers chosen so that each file has a UFID that is unique amongst all of the files in a distributed system. When the flat file service receives a request to create a file it generates a new UFID for it and returns the UFID to the requester.

Directory service: the directory service provides a mapping between *text names* for files and their UFIDs. When a file is created, the current file is created, the client module must record the UFID of

each file in a directory, together with a text name. When a text name for a file has been recorded in this way, clients may subsequently obtain the UFID of the file by quoting its text name to the directory service. The directory service provides the functions needed to generate and update directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service.

Client module: The client module is an extension of the user package. A single client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. The client module also holds information about the network locations of the flat file server and directory server processes.

DESGIN ISSUES

A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance.

Flat file service: Our flat file service model is designed to offer a simple, general-purpose set of operations. Files contain both *data* and *attributes.* The data consist of a sequence of data items, accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. A suitable attribute record structure

The remaining attributes, including the UserID of the file's owner and the access control list are maintained and accessed by the directory service; it would be unnecessarily costly for the flat file service to check user's authorizations before executing every request to access a file. That is the responsibility of the directory service and is performed when ever the directory service processes a client's request for a UFID.

File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

Fault tolerance: The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures. The RPC interfaces can be designed in terms of *idempotent* operations ensuring that duplicated requests do not result invalid updates to files, and the servers can be *stateless*, so that they can be restarted and the services restored after a failure without any need to recover previous state.

Directory service: Directory service that creates and modifies entries in simple one-dimensional directories, looks up text names in directories and returns the corresponding UFID after checking the user's authorization.

The translation from file name to UFID performed by the directory service is a *stateless* substitute for the *open file* operation found in non-distributed systems. The directory service also takes responsibility for access control, and this requires that UFIDs take the role of *capabilities*.

Client module: The client module hides low-level constructs such as the UFIDs used in the RPC interfaces of the flat file service and the directory service from user-level programs, emulating a set of functions similar to the input-output functions of the host operating systems in the client node. When files are located in several nodes, the client module is responsible for locating them, based on the identity of the file's group

INTERFACES

We describe service interfaces by listing their producers, giving a brief explanation of the action of each procedure. We use the following notation for specifying the name of a procedure, its inputs and results, any error conditions that may arise and a description of its operation:

---

procedure name (argument 1, argument2,…) -> (result1,result2,…)-

REPORTS (error1, error2,…)

Description.

The input parameters are listed in brackets after the name of the operation, the names of parameters follow a set of naming conventions defined below. The results are listed after the input parameters, separated from them by an arrow and have names chosen according to the same convention. Any exceptions or error conditions that may arise in a procedure are identified by names listed after the word REPORTS. The following names for parameters and results

File	the UFID of a file
i, n,l	integers
Data	a sequence of data items

Attr	a record containing the attributes of a file
Dir	a UFID referring to directory
Name	a text name
AccessMode	a file service operation for which a UFID is required, for example (read, write, delete …) or a combination of these a regular expression
Pattern	a regular expression
userID	an identifier enabling the directory service to identity a client
BadPosition	error: invalid position in file
NotFound	error: name absent from directory
NoAccess	error: caller does not have access permission
NameDuplicate	error: attempt to add name already in directory

For example, the procedure definitions:

Read (*File, I, n*) -> *Data – REPORTS (*Badposition*)*

Defines the procedure Read with three arguments – the UFID of a file and two integers-and returns a sequence of data items as its result. It will report a BadPosition error if the argument i is outside the bounds of the file.

FLAT FILE SERVICE

This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A UFID is invalid if the file that it refers to is not normally used directly by user-level programs. A UFID is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* report an error if the *File* argument contains an invalid UFID.

The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter *i* specifying a position in the file. The *Read* operation copies the sequence of n data items beginning at item i from the specified file into *Data*, which is then returned to the client. The write operation copies the sequence of data items in data into the specified file beginning at item I, replacing the previous contents of the file at the corresponding position and extending the file if necessary

It is some times necessary to shorten a file; truncate does so. *Create* creates a new, empty file and returns then UFID that is generated. *Delete* removes the specified file. *Get attributes* and *Set attributes* enable clients to access the attribute record. Get attributes is normally available to any client that is allowed to read the file. Access to the Set attributes operation would normally be restricted to the directory service that provides access to the file. The values of the length and timestamp portions of the

attribute record are not affected are not affected by *SetAttributes;* they maintained separately by the flat file service itself.

Read (File, I, n) -> (Data) – REPORTS (Bad Position)

If 1<= I <= length (file):

Reads a sequence of up to n items in file starting at item I and returns it in data.

If I > length (file):

Returns the empty sequence, reports an error.

Write (file, I, data) – REPORTS (bad position)

If 1<=i

Comparison with UNIX: Our interface and the UNIX file system primitives are functionally equivalent. It is simple matter to construct a client module that emulates the UNIX system calls in terms of our flat file service and the directory service operations described in the next section.

In the comparison with the UNIX interface, our flat file service has no *open* and *close* operations – files can be accessed immediately by quoting the appropriate UFID. The Read and Write requests in our interface include a parameter specifying a starting point within the file for each transfer, whereas the equivalent UNIX operations do not. In UNIX, each *read* or *write* operation starts at the current position of the read-write pointer and the read-write pointer is advanced by the number of bytes transformed after each read and write and seek operation is provided to enable the read – write pointer to be explicitly repositioned.

The interface to our flat file service differs from the UNIX file system interface mainly for reasons of fault tolerance:

Repeatable operations: With the exceptions of create, the operations are idempotent, allowing the use of at-least-once RPC semantics – clients may repeat calls to which they receive no reply. Repeated execution of create produces a different new file for each call, causing a space leak, but has no other ill-effects. We shall discuss the implications of the space leak.

Stateless severs: the interface is suitable for implementation by stateless servers can be restarted after a failure and resume operation