CHAPTER 16: DISTRIBUTED-SYSTEM STRUCTURES

- Network-Operating Systems
- Distributed-Operating Systems
- Remote Services
- Robustness
- Design Issues

Network-Operating Systems – users are aware of multiplicity of machines. Access to resources of various machines is done explicitly by:

- Remote logging into the appropriate remote machine.
- Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism.

Distributed-Operating Systems – users not aware of multiplicity of machines. Access to remote resources similar to access to local resources.

- Data Migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task.

- Computation Migration – transfer the computation, rather than the data, across the system.

- Process Migration – execute an entire process, or parts of it, at different sites.
  - Load balancing – distribute processes across network to even the workload.
  - Computation speedup – subprocesses can run concurrently on different sites.
  - Hardware preference – process execution may require specialized processor.
  - Software preference – required software may be available at only a particular site.
  - Data access – run process remotely, rather than transfer all data locally.

Remote Services

- Requests for access to a remote file are delivered to the server. Access requests are translated to messages for the server, and the server replies are packed as messages and sent back to the user.

- A common way to achieve this is via the *Remote Procedure Call* (RPC) paradigm.

- Messages addressed to an RPC daemon listening to a *port* on the remote system contain the name of a process to run and the parameters to pass to that process. The process is executed as requested, and any output is sent back to the requester in a separate message.

- A *port* is a number included at the start of a message packet. A system can have many ports within its one network address to differentiate the network services it supports.

The RPC scheme requires binding client and server port.

- Binding information may be predecided, in the form of fixed port addresses.
  - At compile time, an RPC call has a fixed port number associated with it.
  - Once a program is compiled, the server cannot change the port number of the requested service.

- Binding can be done dynamically by a rendezvous mechanism.
  - Operating system provides a rendezvous daemon on a fixed RPC port.
  - Client then sends a message to the rendezvous daemon requesting the port address of the RPC it needs to execute.

- A distributed file system (DFS) can be implemented as a set of RPC daemons and clients.
  - The messages are addressed to the DFS port on a server on which a file operation is to take place.
  - The message contains the disk operation to be performed (i.e., **read**, **write**, **rename**, **delete**, or **status**).
  - The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client.

Threads

- Threads can send and receive messages while other operations within the task continue asynchronously.

- *Pop-up thread* – created on ‘‘as needed’’ basis to respond to new RPC.
  - Cheaper to start new thread than to restore existing one.
  - No threads block waiting for new work; no context has to be saved, or restored.
  - Incoming RPCs do not have to be copied to a buffer within a server thread.

- RPCs to processes on the same machine as the caller made more lightweight via shared memory between threads in different processes running on same machine.

The DCE thread calls

1. Thread-management:

   create, exit, join, detach

2. Synchronization:

   mutex_init, mutex_destroy, mutex_lock, mutex_trylock, mutex_unlock

3. Condition-variable:

   cond_init, cond_destroy, cond_wait, cond_signal, cond_broadcast

4. Scheduling:

   setscheduler, getscheduler, setprio, getprio

5. Kill-thread:

   cancel, setcancel

Robustness

To ensure that the system is robust, we must :

- *Detect* failures.
  - link
  - site

- *Reconfigure* the system so that computation may continue.

- *Recover* when a site or a link is repaired.

Failure Detection – To detect link and site failure, we use a *handshaking* procedure.

- At fixed intervals, sites A and B send each other an *I-am-up* message. If site *A* does not receive this message within a predetermined time period, it can assume that site *B* has failed, that the link between *A* and *B* has failed, or that the message from *B* has been lost.

- At the time site A sends the *Are-you-up?* message, it specifies a time interval during which it is willing to wait for the reply from *B*. If A does not receive B’s reply message within the time interval, A may conclude that one or more of the following situations has occurred:
  1. Site *B* is down.
  2. The direct link (if one exists) from *A* to *B* is down.
  3. The alternative path from *A* to *B* is down.
  4. The message has been lost.

Reconfiguration – Procedure that allows the system to reconfigure and to continue its normal mode of operation.

- If a direct link from *A* to *B* has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.

- If it is believed that a site has failed (because it can no longer be reached), then every site in the system must be so notified, so that they will no longer attempt to use the services of the failed site.

Recovery from Failure – When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

- Suppose that a link between *A* and *B* has failed. When it is repaired, both *A* and *B* must be notified. We can accomplish this notification by continuously repeating the handshaking procedure.

- Suppose that site *B* has failed. When it recovers, it must notify all other sites that it is up again. Site *B* then may have to receive from the other sites various information to update its local tables.

Design Issues

- Transparency and locality – distributed system should look like conventional, centralized system and not distinguish between local and remote resources.
- User mobility – brings user's environment (i.e., home directory) to wherever the user logs in.
- Fault tolerance – system should continue functioning, perhaps in a degraded form, when faced with various types of failures.
- Scalability – system should adapt to increased service load.
- Large-scale systems – service demand from any system component should be bounded by a constant that is independent of the number of nodes.
- Process structure of the server – servers should operate efficiently in peak periods; use of lightweight processes or threads.