Modularity with Clients and Services

Monday, December 03, 2007
4:19 PM

- Procedure calls.
    - Modules in computer systems (especially software modules) interact by calling others' procedures.
    - Calling procedures provides modularity, but it's *soft modularity*: the modularity is specified by a contract. Nothing is there to enforces it.
    - Consider a normal procedure call. There's a lot of stack juggling underneath:
        - The caller must store arguments on stack.
        - The caller must also store return address on the stack.
        - The caller jumps to the callee.
        - The callee loads the arguments from stack.
        - The callee computes.
        - The callee loads the return address from the stack.
        - The caller jumps to the caller specified return address.
        - The caller resumes control.
    - What if:
        - The callee corrupts the caller's stack area?
        - The callee returns somewhere else besides what the caller specified?
        - The callee goes into an infinite loop? (This disaster is called *fate sharing*.)
        - The callee modifies some global variables it's not supposed to modify?
    - This is not good. We need *enforced modularity*. Modularity that has an external mechanism enforcing it.

- Client/service organization
    - Limit interactions between modules to sending and receiving messages.
    - Service = a module that is used.
    - Client = a module that uses a service.
    - Client & server interaction
        - Client builds a message containing all information the service needs to do its job.
        - Client sends the message to the service.
        - Service extracts argument from the message.
        - Service computes.
        - Service sends the result back to client.
        - Client extracts result from message.
    - Client-server interaction can be represented by *message timing diagram*.
    - In the ideal case, client and service are supposed to be run on separate machines connected by a network.
        - Client and server can really only communicate via message.
        - If they are geographically separated, they are less likely to be subjected to the same disasters such as fire or power outages.
        - However, it is costly to have one computer for each module.
    - Characteristics
        - Errors can propagate only through messages.
        - No global, shared data structures.
        - Clients and servers do not have to rely on one another for correct operations.
        - Clients can put time limit on how long it waits for responses from services.
        - Sweeping generalization: everything through messages.
    - Advantages
        - Enforcing modularity
            - Limit ways that modules can interact. Fewer ways to interact, fewer ways things can go wrong.
        - Fault tolerance
            - If the service fails, the client just loses the function the service provides. It doesn't fail altogether. So it's easy to recover from outages.

                - If modules check messages for validity, they can control how errors propagate somewhat.
            - Security
                - Modules can examine incoming messages for potential attacks.
    - Examples
        - WWW
        - DNS

- Organizations made possible by client-service model
    - Multiple clients and services can work at the same time.
    - A module can be both client and server at the same time.
    - A service may be implemented by multiple identical modules.
        - Good for fault tolerance.
        - Complicates the system somewhat because we have to be able to name each module.
    - Trusted intermediaries
        - A service that functions as a trusted third party for a number of untrusting clients.
        - Examples:
            - File system.
            - Email services.
            - Certificate authorities.
        - Advantages
            - Centralized control over shared resources.
            - An easy way to achieve protection.
            - Enforce modularity between clients. A trusted intermediary makes sure that faults at one client have limited effects on other clients.
            - Client can be simple.
        - Disadvantages
            - The trusted intermediary itself is expensive to design, implement, and maintain.
            - Can become bottleneck.
            - If it fails, all the function is lost.
            - Users have to trust it. What if it's malicious or is censored?
    - Peer-to-peer
        - Every module participating is equal in function, but not capacity. A module is called a node.
        - Every node is both client and service.
        - A node is connected to many other nodes. This makes the system fault tolerance.
        - A node request services it wants from other nodes in the network, while providing services that it is capable of to them at the same time.
        - Information is not centralized.
            - Need distributed algorithm to discover services.
            - For every request, some services are going to be missed.
            - Most design has characteristics that popular services are easier to find.
        - Napster
            - Music sharing network programmed by a 18 year-old boy.
            - Songs are stored on nodes, but a trusted intermediary store their locations.
            - RIAA could shoot Napster down in one shot because of this trusted intermediary.
            - New systems avoid trusted intermediaries to avoid one point failure.

- Remote Procedure Call (RPC)
    - A stylized communication between client and service where every message is followed by a response.
    - With RPC, one can mask client-service communication so that it looks like normal procedure calls.
    - Stub
        - Hides implementation details.
        - Make a message from its argument. (*Marshalling)*

    - Send message, and wait for response.
    - Extract results form the response. (*Unmarshalling)*
  - RPC has different semantics from local procedure call.
    - Caller and callee of RPCs don't share fate.
    - Client must be more complicated because it has to deal with incorrect messages or the lack of responses from service.
  - Three modes of responding to lack of responses from service.
    - At-least-once RPC: The client keeps trying until it gets response. It's important that the computation performed is *idempotent*: the result of performing it more than once is the same at performing it only once.
    - At-most-once RPC: The client just gives up and declare that the RPC fails. This mode is preferable for RPC that have side effects; for example, electronic payment.
    - Exactly-once RPC: The client communicates with the service after it comes back up to see what went wrong and tries to ensure that the computation is performed only once. This is very hard to achieve in real systems.

- Communicating through intermediary
  - Communication between client and service require them to be present at the same time.
  - With trusted intermediary, we can implement *buffered communication*, in which the intermediary holds the message until the receiver is available.
  - Email and newsgroup are prime example of this mode of communication.
  - Styles of communications
    - Push/pull
      - Push = send the message to the intermediary.
      - Pull = receive message from the intermediary.
      - Push and pull operations often have different protocols. For example, Simple Mail Transfer Protocol (SMTP) is a push protocol, while IMAP and POP are pull protocols.
    - Publish/subscribe
      - The *publisher* notifies the intermediary that it has a message on a particular topic.
      - Any user interested in the topic may *subscribe* to that topic, so that when he pulls messages from the intermediary, new messages on the topic are delivered to him.
      - Examples: mailing list, internet chat room.
  - Things you can achieve through intermediary.
    - Indirection
      - The intermediary can send the message to an appropriate user that might not be the one indicated by the name the sender utters.
    - Send the same message to multiple receivers.

- Example: Network File System (NFS)
  - Allow computers to access files across LAN or WAN.
  - Possibly a peer-to-peer architecture. Nodes are computers connected to the network. Each has an independent file system. A node can be both a client (accessing files on other nodes) or a server (storing files locally and allowing other nodes to access the files).
  - Goal: Allow nodes to access file on other nodes using the same interface as the file system interface on the node itself.
  - Two separate protocols:
    - Mounting protocol
    - NFS protocol
  - Mount protocol
    - Establish initial logical connection between server and client.
    - Managed by a process outside a kernel.
    - Client Interface
      - NFS_Mount(local_directory, remote_directory, hostname)
    - Semantics
      - local_directory is replaced by remote_directory. (If you ls local_directory, you'll see files in remote_directory.)

    - Server maintains:
        - An *export list* of directories it allows clients to mount.
        - A list of names of machines that are allowed to mount those directories.
    - When server receives a mount request, it checks the request with the export list. After clearing permission, it returns a file handle to the client. In Unix, the file handle consists of the name of the file system being mounted, and the inode number of the directory being mounted.
- NFS Protocol
    - Protocol for accessing files.
        - Manipulating links and directories.
        - Reading and writing files.
    - Implemented as RPCs.
    - No open() and close(). Because NFS wants to be *stateless*. NFS doesn't want to keep any information about the state of the clients, so that it doesn't have to recover those states when it fails. This makes NFS more robust.
    - File operations are *idempotent*. So, client can deal with lack of response using more-than-once RPC.
    - However, each NFS request has a request number, so that the server can see which request is duplicated or lost.
    - Stateless server dictates that each write must be committed before returning to the client. This makes caching harder.
    - A single write RPC is atomic. However, a write() system call at the client can be broken down into a number of RPCs. So, if two users concurrently write to the same file, they data may get mingled.
    - NFS does not provide locking because it is stateful. Users are encouraged to coordinate writes to the same file themselves.
- Path name resolution
    - Path names have components.
    - Each component is resolved by separate NFS lookup RPC once a mount point is crossed.
    - Have to do this because some implements allow cascading mounts.
        - Say /nfs/theory may be mounted to a directory on theory.cpe.ku.ac.th.
        - But user might mount /nfs/theory/etc/practice to a directory on practice.cpe.ku.ac.th
        - Thus, can't just hand etc/practice to theory.cpe.ku.ac.th to resolve.
    - Server can't act as intermediary between client and a server it is using service.
        - nagato.cpe.ku.ac.th mounts /nfs/theory to /home/pramook on theory.cpe.ku.ac.th.
        - /home/pramook/etc/practice is a directory on theory.cpe.ku.ac.th. The directory has file a.txt.
        - Say, theory.cpe.ku.ac.th mounts /home/pong on practice.cpe.ku.ac.th to /home/pramook/etc/practice. /home/pong has one file, b.txt.
        - When I list /nfs/theory/etc/practice on nagato.cpe.ku.ac.th, I'll see a.txt. However, when I list /home/pramook/etc/practice on theory.cpe.ku.ac.th, I'll see b.txt.
- Performance tuning and its complication
    - Buffering and caching is done on the client side.
        - File attribute cache
            - Record expires every 60 seconds.
            - Record is updated if the information on the server doesn't match.
        - File block cache
            - Client writes locally before flushing the blocks to servers.
    - Consistency semantics is in jeopardy.
        - A file create on one computer might not be visible elsewhere for a while (1-2 minutes). The same is true for writes.
        - Changes are visible only after they are committed to the disks on the server, and clients can see the change only after they update their cache.
- All in all, people still use it because of performance and robustness.
- Worst is better, after all.