Middleware for Communications

Contents:

- Programming Abstractions
- Roles of MW
- Communication Primitives
- RPC Model

Mariano Cilia / cilia@informatik.tu-darmstadt.de

Programming Abstractions

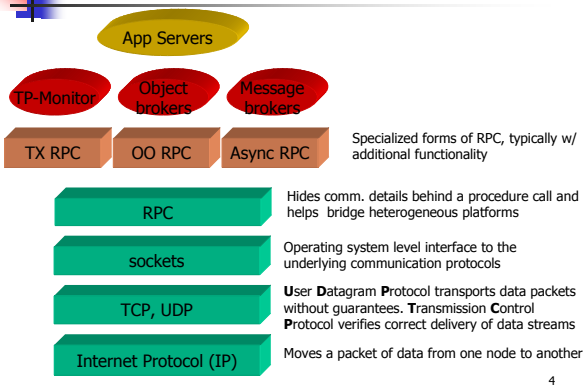
- Programming languages (and almost any form of software system) evolve towards higher levels of abstraction
  - Hiding HW and platform details
  - More powerful primitives and interfaces
  - Leaving difficult task to intermediaries
    - Compilers, optimizers, automatic load balancers, automatic data partitioning, …
  - Reducing the number of programming errors
  - Reducing the development and maintenance cost of the apps
    - Facilitating app portability

[Alonso '04]

Programming Abstractions (cont)

- MW is primarily a set of programming abstractions developed to facilitate the development of complex distributed sys
  - To understand a MW platform one needs to understand its programming model
  - From the programming model the limitations, general performance, and applicability of a given type of MW can be determined in a first approximation
  - The underlying programming model also determines how the platform will evolve and fare when new technologies evolve

The Genealogy of Middleware



Infrastructure

- As the programming abstractions reach higher and higher levels, the underlying infrastructure implementing the abstractions must grow accordingly
  - Additional functionality is almost always implemented through additional SW layers
  - These layers increase the size and complexity of the infrastructure necessary to use the new abstractions

Infrastructure (cont)

- The infrastructure is also intended to support additional functionality that makes development, maintenance, and monitoring easier and less costly
  - RPC → transactional RPC → logging, recovery, advanced TX models, language primitives for TX'nal demarcation, …
  - The infrastructure is also there to take care of all the non-functional properties typical ignored by data models, programming models, and programming langs: performance, availability, recovery, maintenance, resource mgmt, …

Understanding Middleware

To understand MW, it is needed to understand its dual role:

- **Prog. Abstraction**
  - Intended to hide low level details of HW, networks and distribution
  - Trend is towards increasingly more powerful primitives that, without changing the basic concept of RPC, have additional properties or allow more flexibility in the use of the concept
  - Evolution and appearance to the programmer is dictated by the trends in programming languages
    - RPC and C; RMI and Java, Web services and SOAP
- **Infrastructure**
  - Intended to provide a comprehensive platform for developing and running complex distributed systems
  - Trend is towards SOAs at a global scale and standardization of interfaces
  - Evolution is towards integration of platforms and flexibility in the configuration (plus autonomic behavior)

Interaction between C and S

- Imagine we have a program that implements certain services (S).
- Imagine we have other programs (C) that would like to invoke those services
- To make the problem more interesting, assume as well that:
  - C and S can reside on different computers
    - running different OSs
  - The only form of communication is by sending messages (no shared memory, no shared disks, …)
- We want a generic solution and not a one-time-hack
  - One cannot expect the programmer to implement a complete infrastructure for every distributed application

Problems to Solve

- How to make the service invocation part of the prog. language in a "transparent" manner?
- How to exchange data between machines that might use different representations for data types?
  - Data type formats (byte order in different architectures)
  - Data structures (need to be flattened and reconstructed)
- How to find the service among a potentially large collection of services and servers?
  - Goal: the client does not necessarily need to know where the server resides or even which server provides the service
- How to deal with errors in the service invocation (in a elegant manner)?
  - Server or communication is down, server is busy, …

Programming Languages

- The notion of distributed service invocation became a reality at the beginning of the 80's (C as prog. lang)
- In procedural languages the basic module is the procedure
  - A procedure implements a particular function or service that can be used anywhere within the program
- It seemed natural to maintain this same notion
  - The client makes a procedure call to a procedure that is implemented by the server.
  - Since the client and server can be in different machines, the procedure is remote
- Several aspects are immediately determined
  - Data exchange is done as input and output parameters of the procedure call
  - Pointers cannot be passed as parameters in RPC

Infrastructure: Interoperability

- Consider data exchanged between clients and servers residing in different environments (HW or SW)
  - Byte order: differences between little and big endian architecture
  - Data structures: like trees, hash tables, records, … need to be flattened before being sent
- This is best done using an intermediate representation format, known as
  - Marshaling/un-marshaling
  - Serializing/de-serializing
- Having an intermediate representation simplifies the design
  - Otherwise a node will need to be able to transform data to any possible format

Basic Comm. Middleware: RPC

- What does RPC?
  - Hides distribution behind procedure calls
  - Provides an interface definition language (IDL) to describe services
  - Generates all the additional code necessary to make a procedure call remote and to deal with all the communication aspects
  - Provides a binder in case it has a distributed name and directory service

Conventional Comm. MW Today

- RPC and the model behind RPC are at the core of any MW platform
  - even those using asynchronous interactions
- RPC, however, has become part of the low level infrastructure and it is rarely used directly by application developers

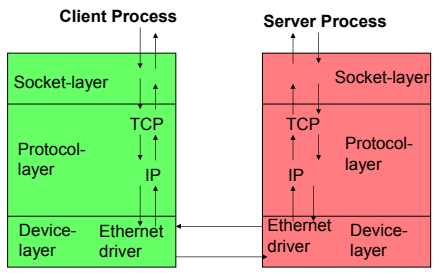
Communication Primitives

- Sockets
- RPC
- Group Communication

Sockets

- Typical low-level protocol in Unix environments
- Bidirectional communication channel
- Three parts visible to user:
  - socket layer
  - protocol layer
  - device layer
- At system generation-time the valid combinations of socket, protocol and device are specified

Sockets



Sockets

- Depending on the protocol, two possible communication models:
  - stream: virtual circuit between processes, messages sent (*send*) and received (*recv*)
  - datagram: message sent to one or more recipients (*sendto*) and received from them (*recvfrom*)
- No guaranteed reception or ordering
- Special code needed for error handling

RPC Model

- Remote procedure calls mimic behavior of local procedure calls for communication among different/remote processes
- Synchronous communication
  - calling process stops, waits for procedure to execute and control to be returned

Advantages of RPC

- Programming can ignore distribution
- Remote calls treated "same" as local calls
- Safe: every call receives exactly one return
  - return from called procedure or
  - exception from exception handler
- No sequencing of messages required
- Language transparency for parameters
- HW transparency

19

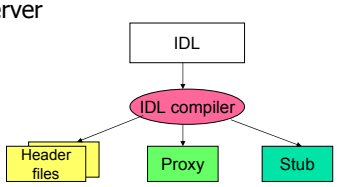
Binding Clients and Servers

- Remote procedure must be referenced at compile time
- Write interface definition for called program
- Interface definition specifies name, type and arguments of procedure
- Interface definition processed by interface definition language (IDL) compiler (stub compiler)

20

Binding Clients and Servers (2)

- IDL compiler produces
  - header files to be included in calling program
  - proxy procedures that are linked with caller
  - stub procedures that are linked with the server



21

Marshaling and Unmarshaling

- Proxy packs procedure name and parameters into a stream that can be sent in a message
- Proxy translates parameters to standard format understandable by server
- Stub translates standard format into server's format (e.g. if different data representations are used)
- Alternative: proxy sends as is but tags with format, stub converts if format of client different (receiver-makes-right)

22

Communication Binding

- Communication binding between client and server must be established
- Server must export/register its interface
  - what interface is supported
  - where it can be located
- Client creates communication based on exported interface
- But clients need to know or find the service in question

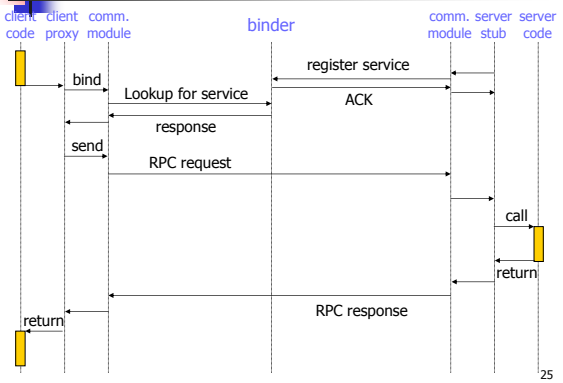
23

Finding / Binding

- A service is provided by a server, listening to a given port
- Binding is the process of mapping
  - service name → address and port
- Binding can be done
  - Locally: client knows the address of the server
  - Distributed: there is a directory service (Binder) in charge of doing the mapping
    - It must be reachable by all participants
      - Ways to locate the binder: predefined location, environment variables, broadcast, …)

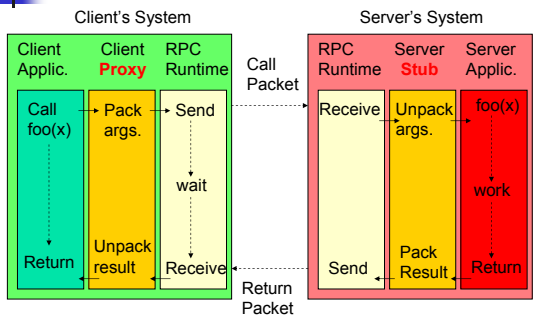
24

Sequence Diagram



25

RPC Walk-through



26

Programming issues when using RPC

- Write interface definition for servers
- Multithreaded client so that blocking a caller doesn't stall client process
- Multithreaded server to support many concurrent clients
- Binding of programs, exporting/importing interfaces
- Lost messages

27

What can go wrong here?

- RPC is a point-to-point protocol in the sense that it supports the interaction between two entities
- When there are more entities interacting with each other, RPC treats the calls as independent of each other (however, these calls are not independent)
- Recovering from partial system failures is very complex
- Avoiding these problems using plain RPC is very cumbersome

28

Fault tolerance in RPC

- Idempotent vs. non-idempotent servers
- If client doesn't receive answer, was procedure call lost or return message?
  - Resend to idempotent server
  - Inquire from non-idempotent server
- Execute
  - maybe (no guarantee)
  - at least once (idempotent operation)
  - at most once (non-idempotent operation)
  - exactly once (committing transaction)

29

Transactional RPC

- Coupling of RPC with transactions
  - transactional messages between C & S
  - atomic procedures (all or nothing)
  - results are made visible through commit
- Additional data must be passed
  - transaction ID
  - coordination information (2 phase commit)
  - name and location of calling TP system
  - exception codes

30

Performance of RPC

- 3 main components
  - marshaling/unmarshaling of parameters
  - RPC runtime and communication software
  - physical network transfer
- Total cost of one RPC ~ 10 000 -15 000 machine instructions
- Local PC approx. 100 times faster!

32

Unicast vs. Broadcast vs. Multicast

- Unicast is unidirectional 1:1 messaging
- Broadcast sends message to all participants
  - e.g. Ethernet
- Multicast sends same message to a group of participants
  - identification of group membership based on
    - physical addresses
    - content of message (subject-based addressing)

33

Unicast

- Unidirectional point to point data transfer
- Control information may flow in both directions (acks, nacks)
  - letter is data transfer
    - if sent with return receipt, receipt is just control info but not a new data transfer
  - answer letter is new data transfer
- Unicast doesn't scale for large groups

34

Group Communication: Multicast

- Same message sent by one process to a group of processes
- Useful infrastructure for fault tolerance in distributed applications
- Different options (quality of service):
  - unreliable multicast
  - reliable multicast
  - atomic multicast

35

Unreliable Multicast

- Best effort message delivery
- No guarantees are given
- Acceptable mostly for non-critical messages
  - load info
- Used by some data dissemination applications

36

(k)-Reliable Multicast

- Reliable multicast provides guaranteed delivery to (at least k) receivers
- Reliability depends on failure model
- As k grows, cost increases
- Different qualities of service possible

37

Atomic Multicast

- Message is received by all members of group or by none
- Failed processes are excluded from a group

38

Ordering in Multicast

- Ordering specifies the sequence in which messages are sent and delivered/received
  - FIFO (based on single sender)
  - total ordering (requires centralized control)
  - causal ordering (establishes partial orders over mutually relevant messages)

39

Point-to-Point Messaging

- Processes communicate through explicit messages
- No master/slave relationship but equals
- More flexibility than RPC in terms of allowable message sequences and termination sequencing
- Higher programming complexity

40

RPC vs. Point-to-Point: Connection

- Peer-to-Peer connection oriented:
  - direction of communication
  - direction of the link (i.e. send or receive mode)
  - ID and state of the transaction (active, committed, aborted) and of individual programs (e.g. cursor)
  - connection state not recoverable
- RPC connectionless communication model:
  - client and server don't share state
  - explicit passing of state back and forth
  - context handle provided by some implementations

41

RPC vs. Point-to-Point: Programming

- RPC
  - much simpler
  - same call-return semantics in local and remote procedure calls
  - hard-wired termination, simple semantics
- Point-to-Point
  - communication mechanism reflected in application logic
  - timing information (waiting for messages) part of program

42