title: Models notebook: Distributed Systems layout: note date: 2020-08-12 tags: …

## Models

Models are used to provide abstract, simplified, consistent description of some aspect of interest of distributed system design.

- **physical**: describe types of computers/devices that constitute a system and their interconnectivity without details of specific computer/networking technologies
  - most explicit
- **architectural**: describe system in terms of computational and communication tasks performed by computational elements
  - e.g. client-server, peer-to-peer
- **fundamental**: abstract perspective to describe solutions to particular issues faced by most distributed systems
  - e.g. interaction, failure, security

## Physical Models

- **baseline**: extensible set of computer nodes interconnected by computer network for passing of messages
- 3 generations of distributed systems

### 1970s-80s, early distributed systems

- 10-100 nodes
- local area network, usually Ethernet
- limited Internet connectivity: file transfer, email
- shared local printers and file servers

#### 1990s, Internet-scale distributed systems

- large-scale distributed systems emerged with rapid growth in Internet
- extensible set of nodes interconnected by network of networks (i.e. the Internet)
- significant heterogeneity: networks, computer architecture, operating systems, languages, …
- emphasis on open standards

#### Contemporary distributed systems

- nodes in earlier generations were primarily desktop computers which were:
  - *static*: not moving around
  - *discrete*: not embedded in other objects
  - *autonomous*: largely independent of other computers
- in contrast, modern distributed systems don't satisfy these properties:
  - *mobile computing*: nodes have varying location, needing service discovery and spontaneous interoperation
  - *ubiquitous computing*: computers embedded in everyday objects
  - *cloud computing*: pools of nodes together providing a service

### Architectural Models

- system architecture: structure in terms of separately specified components and their interrelationships
  - goal: ensure the structure meets present/future demand
  - concerns: reliability, managability, adaptability, cost-effectiveness
- **architectural elements**: interacting components of system
- **architectural patterns**:
- **middleware**

#### Architectural Elements

Building blocks:

- communicating entities: e.g. threads/processes, nodes
- communication paradigm: e.g. message queue, publish/subscribe

- roles and responsibilities: e.g. client, server, peer
- placement: mapping onto physical distributed infrastructure

### Communicating Entities

System-oriented perspective

- **processes**: usually processes are the communicating entities
  - **threads**: strictly threads may be the endpoints of communication
- **nodes**: sensor networks, OS may not support process abstraction

Problem-oriented perspective

- **objects**: objects accessed via interfaces
- **components**: specify interfaces and make dependencies explicit, providing a more complete contract with which to construct the system than objects
- **web services**: closely related to objects/components. Intrinsically integrated with the WWW, using web standards to represent and discover services.
  - software application identified by a URI with interfaces defined, described, discovered as XML
  - supports direct interaction with other software agents via XML message exchange through IP
- objects/components are usually internal to an organisation for tightly coupled applications
- web services are complete services

### Communication Paradigms

#### Direct: coupled senders/receivers

- **interprocess communication**: low-level support for communication between processes in distributed systems
  - message passing primitives, socket programming, multicast
- **remote invocation**: most common communication paradigm for distributed systems
  - 2 way exchange between communicating entities which resulting in remote operation being called
- **request-reply protocol**: pattern on message-passing to support client-server computing

- pairwise message exchange
- most DS use RPC/RMI, but both are supported by underlying request-reply exchanges

- **remote procedure call (RPC)**: procedures in processes on remote computers can be called as if they are procedures in the local address space.
  - baked in access and location transparency
- **remote method invocation (RMI)**: resembles RPC but in a world of distributed objects
  - a calling object invokes a method in remote object

**Indirect: allow decoupling of senders/receivers** Uncoupling

- **space uncoupling**: senders don't need to know who they are sending to
- **time uncoupling**: senders/receivers don't need to exist at the same time

Techniques

- **group communication**: delivery of messages to set of recipients; one-to-many
  - abstraction of group with an ID, which maintains group membership
  - recipients elect to receive messages by joining a group
  - senders send messages to the group using the group ID
- **publish-subscribe**: large number of producers distributing information to a large number of consumers (with different interests); one-to-many
  - uses intermediary service to ensure efficient routing of information from producers to consumers
- **message queues**: point-to-point service; producer sends messages to a specified queue
  - consumer receives messages from the queue
- **tuple space**: processes can place structured data in a persistent tuple space
  - other processes can read/remove tuples by specifying patterns of interest
  - readers/writers don't need to exist simultaneously
- **distributed shared memory**: abstraction for sharing data between processes that don't share physical memory

**Roles and Responsibilities**

- **client-server**: client processes interact with individual server processes
  - client processes establish connections
  - server processes listen for incoming connections
  - most important and most widely used architecture
- **peer-to-peer**: all processes play similar role as peers
  - resources of each peer are used, so the system resources scale with the number of users
  - substantially more complex than client-server architecture
  - e.g. BitTorrent

**Placement**

- how objects/services map onto physical distributed infrastructure
- crucial determinant of DS properties: performance, reliability, security
- where to place a given client/server in terms of machines/processes
- needs to account for e.g.:
  - communication pattern between entities
  - reliability of machines and current loading
  - quality of communications
- **mapping services to multiple servers**: services can be implemented as multiple server processes in separate host computers which interact to provide a service to clients
  - can partition/replicate across multiple hosts
  - **cluster**: thousands of commodity processing boards
- **cache**: store of recently used objects that is closer to some clients than objects themselves
  - reduce network traffic and server load
  - may improve performance for client
- **proxy server**: increase availability/performance by reducing load on wide area network
- **mobile code**: applets/Javascript
  - good interactivity, allows asynchronous behaviour (AJAX)
  - server provides code to browser which client browser runs to access services
  - potential security threat

- **mobile agent**: running program (code + data) that travels from computer to computer, carrying out a task on someones behalf, typically collecting data, eventually returning with results
  - reduced communication cost and time by replacing remote invocations with local ones
  - big security threat, and web crawlers can still access resources successfully through remote invocations
  - used to install/maintain software on computers within organisation

**Architectural Patterns**

- patterns build on primitive architectural elements, providing composite recurring structures that work well in particular circumstances
- **layering**: partition system into layers, with a given layer making use of services provided by the layer below. Higher layers are unaware of lower layer implementation details
  - vertical organisation of services into service layers
  - platform: lowest-level hardware/software; e.g. Intel x86/Linux
  - middleware: software that masks heterogeneity and provides useful programming model. Processes interact to implement communication/resource-sharing. Provides building blocks for constructing software
- **tiered architecture**: complements layering. Horizontal partitioning within a layer, separating functionality into different servers. e.g.
  - presentation logic: user interaction/view as presented to user
  - application logic: app-specific processing/business logic
  - data logic: persistet storage; DBMS
  - 3-tier: separates each logical element into a distinct server
- **thin client**: software layer supportig window-based UI local to the user while accessing services on a remote computer
  - allows simple, low-cost devices to be used with a wide range of services
  - poor performance for highly interactive graphical activities: CAD, image processing
  - **Virtual Network Computing (VNC)**: remote access to GUI through VNC client via VNC protocol
- **proxy**: support location transparency in RPC/RMI
  - proxy created in local address space to represent remote object: offers same interface as remote object, meaning application programmer calls on the proxy without knowing about the distributed nature

- also used for replication/caching

- **brokerage**: supports interoperability in complex distributed systems

## Fundamental Models

- should only contain essential ingredients to understand/reason about aspect's of system behaviour
- make explicit relevant assumptions
- make generalisations concerning what is possible/impossible given those assumptions

### Interaction Models

Message passing between processes produces

- communication: information flow
- coordination: synchronisation, ordering

### Failure Models

Define and classifies faults

### Security Models

## Questions

2.1 Provide three specific and contrasting examples of the increasing levels of heterogeneity experienced in contemporary distributed systems as defined in Section 2.2. page 39

- ubiquitous computing: e.g. smart fridges, mobile phones, tablets, laptops, … with significant differences in performance, input devices, etc.
- mobile computing: nodes may move from location to location
- cloud computing: pools of nodes that together provide a service

2.2 What problems do you foresee in the direct coupling between communicating entities that is implicit in remote invocation approaches? Consequently, what advantages do you anticipate from a level of decoupling as offered by space and time uncoupling? Note: you might want to revisit this answer after reading Chapters 5 and 6. page 43

- idempotency

2.3 Describe and illustrate the client-server architecture of one or more major Internet applications (for example, the Web, email or netnews). page 46

the Web: a client (i.e. a browser) opens TCP connections in order to send HTTP requests. A Web server, listening on port 80, responds with a HTTP response.

2.4 For the applications discussed in Exercise 2.1, what placement strategies are employed in implementing the associated services? page 48