Distributed Systems

(3rd Edition)

Chapter 01: Introduction

Version: August 29, 2017

Distributed System

Definition

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

Characteristic features

- Autonomous computing elements, also referred to as nodes, be they hardware devices or software processes.
- Single coherent system: users or applications perceive a single system $\Rightarrow$ nodes need to collaborate.

Collection of autonomous nodes

Independent behavior

Each node is autonomous and will thus have its own notion of time: there is no global clock. Leads to fundamental synchronization and coordination problems.

Collection of nodes

- How to manage group membership?
- How to know that you are indeed communicating with an authorized (non)member?

Organization

Overlay network

Each node in the collection communicates only with other nodes in the system, its neighbors. The set of neighbors may be dynamic, or may even be known only implicitly (i.e., requires a lookup).

Overlay types

Well-known example of overlay networks: peer-to-peer systems.

Structured: each node has a well-defined set of neighbors with whom it can communicate (tree, ring).

Unstructured: each node has references to randomly selected other nodes from the system.

Coherent system

Essence

The collection of nodes as a whole operates the same, no matter where, when, and how interaction between a user and the system takes place.

Examples

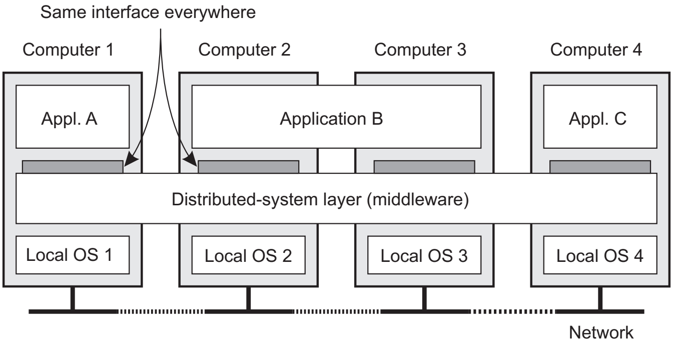
- An end user cannot tell where a computation is taking place
- Where data is exactly stored should be irrelevant to an application
- If or not data has been replicated is completely hidden

Keyword is distribution transparency

The snag: partial failures

It is inevitable that at any time only a part of the distributed system fails. Hiding partial failures and their recovery is often very difficult and in general impossible to hide.

Middleware: the OS of distributed systems



What does it contain?

Commonly used components and functions that need not be implemented by applications separately.

What do we want to achieve?

- Support sharing of resources
- Distribution transparency
- Openness
- Scalability

Sharing resources

Canonical examples

- Cloud-based shared storage and files
- Peer-to-peer assisted multimedia streaming
- Shared mail services (think of outsourced mail systems)
- Shared Web hosting (think of content distribution networks)

Observation

“The network is the computer”

(quote from John Gage, then at Sun Microsystems)

Distribution transparency

Types

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Degree of transparency

Observation

Aiming at full distribution transparency may be too much:

- There are communication latencies that cannot be hidden
- Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash
- Full transparency will cost performance, exposing distribution of the system
  - Keeping replicas exactly up-to-date with the master takes time
  - Immediately flushing write operations to disk for fault tolerance

Degree of transparency

Exposing distribution may be good

- Making use of location-based services (finding your nearby friends)
- When dealing with users in different time zones
- When it makes it easier for a user to understand what's going on (when e.g., a server does not respond for a long time, report it as failing).

Conclusion

Distribution transparency is a nice a goal, but achieving it is a different story, and it should often not even be aimed at.

Openness of distributed systems

What are we talking about?

Be able to interact with services from other open systems, irrespective of the underlying environment:

- Systems should conform to well-defined interfaces
- Systems should easily interoperate
- Systems should support portability of applications
- Systems should be easily extensible

Policies versus mechanisms

Implementing openness: policies

- What level of consistency do we require for client-cached data?
- Which operations do we allow downloaded code to perform?
- Which QoS requirements do we adjust in the face of varying bandwidth?
- What level of secrecy do we require for communication?

Implementing openness: mechanisms

- Allow (dynamic) setting of caching policies
- Support different levels of trust for mobile code
- Provide adjustable QoS parameters per data stream
- Offer different encryption algorithms

On strict separation

Observation

The stricter the separation between policy and mechanism, the more we need to make ensure proper mechanisms, potentially leading to many configuration parameters and complex management.

Finding a balance

Hard coding policies often simplifies management and reduces complexity at the price of less flexibility. There is no obvious solution.

Scale in distributed systems

Observation

Many developers of modern distributed systems easily use the adjective "scalable" without making clear why their system actually scales.

At least three components

- Number of users and/or processes (size scalability)
- Maximum distance between nodes (geographical scalability)
- Number of administrative domains (administrative scalability)

Observation

Most systems account only, to a certain extent, for size scalability. Often a solution: multiple powerful servers operating independently in parallel. Today, the challenge still lies in geographical and administrative scalability.

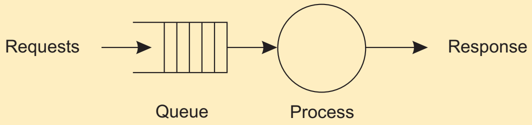
Size scalability

Root causes for scalability problems with centralized solutions

- The computational capacity, limited by the CPUs
- The storage capacity, including the transfer rate between CPUs and disks
- The network between the user and the centralized service

Formal analysis

A centralized service can be modeled as a simple queuing system



Assumptions and notations

- The queue has infinite capacity $\Rightarrow$ arrival rate of requests is not influenced by current queue length or what is being processed.
- Arrival rate requests: λ
- Processing capacity service: μ requests per second

Fraction of time having k requests in the system

$$p_k = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^k$$

Formal analysis

Utilization *U* of a service is the fraction of time that it is busy

$$U = \sum_{k>0} p_k = 1 - p_0 = \frac{\lambda}{\mu} \Rightarrow p_k = (1-U)U^k$$

Average number of requests in the system

$$\overline{N} = \sum_{k\geq 0} k \cdot p_k = \sum_{k\geq 0} k \cdot (1-U)U^k = (1-U)\sum_{k\geq 0} k \cdot U^k = \frac{(1-U)U}{(1-U)^2} = \frac{U}{1-U}$$

Average throughput

$$X = \underbrace{U \cdot \mu}_{\text{server at work}} + \underbrace{(1-U) \cdot 0}_{\text{server idle}} = \frac{\lambda}{\mu} \cdot \mu = \lambda$$

Formal analysis

Response time: total time take to process a request after submission

$$R = \frac{\overline{N}}{X} = \frac{S}{1-U} \Rightarrow \frac{R}{S} = \frac{1}{1-U}$$

with $S = \frac{1}{\mu}$ being the service time.

Observations

- If U is small, response-to-service time is close to 1: a request is immediately processed
- If U goes up to 1, the system comes to a grinding halt. Solution: decrease S.

Problems with geographical scalability

- Cannot simply go from LAN to WAN: many distributed systems assume synchronous client-server interactions: client sends request and waits for an answer. Latency may easily prohibit this scheme.
- WAN links are often inherently unreliable: simply moving streaming video from LAN to WAN is bound to fail.
- Lack of multipoint communication, so that a simple search broadcast cannot be deployed. Solution is to develop separate naming and directory services (having their own scalability problems).

Problems with administrative scalability

Essence

Conflicting policies concerning usage (and thus payment), management, and security

Examples

- Computational grids: share expensive resources between different domains.
- Shared equipment: how to control, manage, and use a shared radio telescope constructed as large-scale shared sensor network?

Exception: several peer-to-peer networks

- File-sharing systems (based, e.g., on BitTorrent)
- Peer-to-peer telephony (Skype)
- Peer-assisted audio streaming (Spotify)

Note: end users collaborate and not administrative entities.

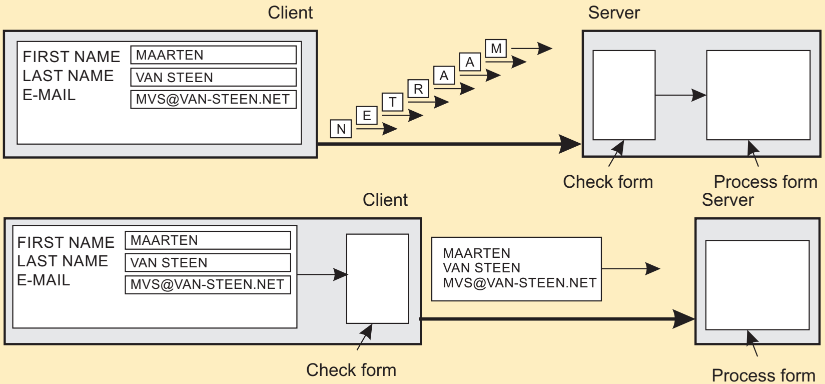
Techniques for scaling

Hide communication latencies

- Make use of asynchronous communication
- Have separate handler for incoming response
- Problem: not every application fits this model

Techniques for scaling

Facilitate solution by moving computations to client



Techniques for scaling

Partition data and computations across multiple machines

- Move computations to clients (Java applets)
- Decentralized naming services (DNS)
- Decentralized information systems (WWW)

Techniques for scaling

Replication and caching: Make copies of data available at different machines

- Replicated file servers and databases
- Mirrored Web sites
- Web caches (in browsers and proxies)
- File caching (at server and client)

Scaling: The problem with replication

Applying replication is easy, except for one thing

- Having multiple copies (cached or replicated), leads to inconsistencies: modifying one copy makes that copy different from the rest.
- Always keeping copies consistent and in a general way requires global synchronization on each modification.
- Global synchronization precludes large-scale solutions.

Observation

If we can tolerate inconsistencies, we may reduce the need for global synchronization, but tolerating inconsistencies is application dependent.

Developing distributed systems: Pitfalls

Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. Many false assumptions are often made.

False (and often hidden) assumptions

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

Three types of distributed systems

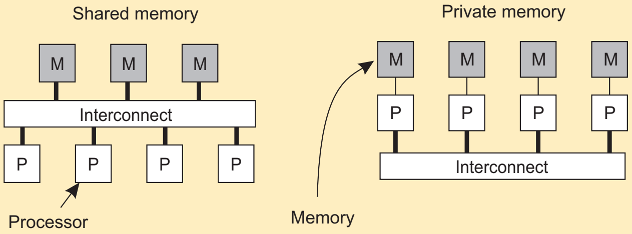
- High performance distributed computing systems
- Distributed information systems
- Distributed systems for pervasive computing

Parallel computing

Observation

High-performance distributed computing started with parallel computing

Multiprocessor and multicore versus multicomputer



Distributed shared memory systems

Observation

Multiprocessors are relatively easy to program in comparison to multicomputers, yet have problems when increasing the number of processors (or cores). Solution: Try to implement a shared-memory model on top of a multicomputer.

Example through virtual-memory techniques

Map all main-memory pages (from different processors) into one single virtual address space. If process at processor *A* addresses a page *P* located at processor *B*, the OS at *A* traps and fetches *P* from *B*, just as it would if *P* had been located on local disk.

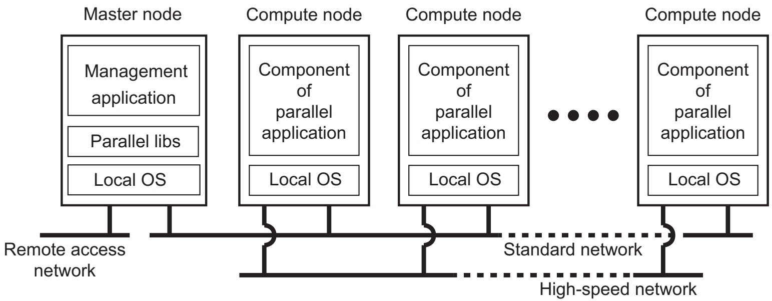
Problem

Performance of distributed shared memory could never compete with that of multiprocessors, and failed to meet the expectations of programmers. It has been widely abandoned by now.

Cluster computing

Essentially a group of high-end systems connected through a LAN

- Homogeneous: same OS, near-identical hardware
- Single managing node



Grid computing

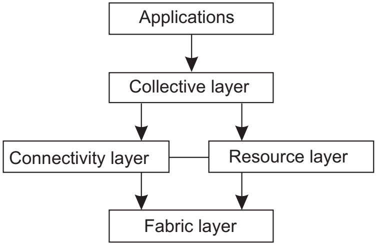
The next step: lots of nodes from everywhere

- Heterogeneous
- Dispersed across several organizations
- Can easily span a wide-area network

Note

To allow for collaborations, grids generally use virtual organizations. In essence, this is a grouping of users (or better: their IDs) that will allow for authorization on resource allocation.

Architecture for grid computing



The layers

Fabric: Provides interfaces to local resources (for querying state and capabilities, locking, etc.)

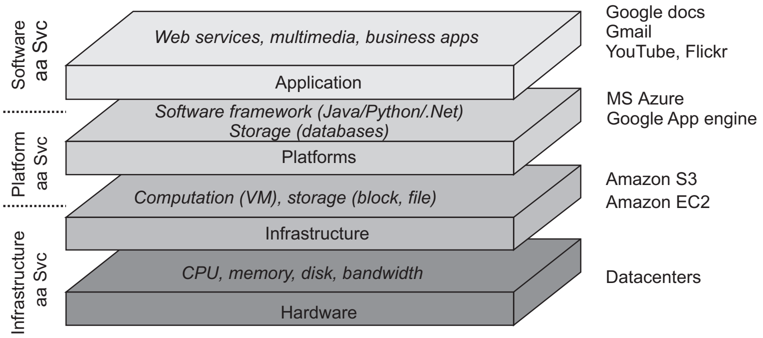
Connectivity: Communication/transaction protocols, e.g., for moving data between resources. Also various authentication protocols.

Resource: Manages a single resource, such as creating processes or reading data.

Collective: Handles access to multiple resources: discovery, scheduling, replication.

Application: Contains actual grid applications in a single organization.

Cloud computing



Cloud computing

Make a distinction between four layers

- Hardware: Processors, routers, power and cooling systems. Customers normally never get to see these.
- Infrastructure: Deploys virtualization techniques. Evolves around allocating and managing virtual storage devices and virtual servers.
- Platform: Provides higher-level abstractions for storage and such. Example: Amazon S3 storage system offers an API for (locally created) files to be organized and stored in so-called buckets.
- Application: Actual applications, such as office suites (text processors, spreadsheet applications, presentation applications). Comparable to the suite of apps shipped with OSes.

Is cloud computing cost-effective?

Observation

An important reason for the success of cloud computing is that it allows organizations to outsource their IT infrastructure: hardware and software. Essential question: is outsourcing also cheaper?

Approach

- Consider enterprise applications, modeled as a collection of components, each component C_i requiring N_i servers.
- Application now becomes a directed graph, with a vertex representing a component, and an arc $\langle \overrightarrow{i,j} \rangle$ representing data flowing from C_i to C_j.
- Two associated weights per arc:
  - $T_{i,j}$ is the number of transactions per time unit that causes a data flow from C_i to C_j.
  - $S_{i,j}$ is the total amount of data associated with $T_{i,j}$.

Is cloud computing cost-effective?

Migration plan

Figure out for each component C_i, how many n_i of its N_i servers should migrate, such that the monetary benefits reduced by additional costs for Internet communication, are maximal.

Requirements migration plan

1. Policy constraints are met.
2. Additional latencies do not violate specific delay constraints.
3. All transactions continue to operate correctly; requests or data are not lost during a transaction.

Computing benefits

Monetary savings

- B_c: benefits of migrating a compute-intensive component
- M_c: total number of migrated compute-intensive components
- B_s: benefits of migrating a storage-intensive component
- M_s: total number of migrated storage-intensive components

Obviously, total benefits are: $B_c \cdot M_c + B_s \cdot M_s$

Internet costs

Traffic to/from the cloud

$$Tr_{local,inet} = \sum_{C_i} (T_{user,i} S_{user,i} + T_{i,user} S_{i,user})$$

- $T_{user,i}$: transaction per time unit causing data flow from user to C_i
- $S_{user,i}$: amount of data associated with $T_{user,i}$

Rate of transactions after migration

Some notations

- $C_{i,local}$: set of servers of C_i that continue locally.
- $C_{i,cloud}$: set of servers of C_i that are placed in the cloud.
- Assume traffic distribution is the same for local and cloud server

Note that $|C_{i,cloud}| = n_i$. Let $f_i = n_i/N_i$, and s_i a server of C_i.

$$T^*_{i,j} = \begin{cases} (1-f_i)\cdot(1-f_j)\cdot T_{i,j} & \text{when } s_i \in C_{i,local} \text{ and } s_j \in C_{j,local} \\ (1-f_i)\cdot f_j \cdot T_{i,j} & \text{when } s_i \in C_{i,local} \text{ and } s_j \in C_{j,cloud} \\ f_i \cdot (1-f_j)\cdot T_{i,j} & \text{when } s_i \in C_{i,cloud} \text{ and } s_j \in C_{j,local} \\ f_i \cdot f_j \cdot T_{i,j} & \text{when } s_i \in C_{i,cloud} \text{ and } s_j \in C_{j,cloud} \end{cases}$$

Overall Internet costs

Notations

- $cost_{local,inet}$: per unit Internet costs to local part
- $cost_{cloud,inet}$: per unit Internet costs to cloud

Costs and traffic before and after migration

$$\begin{aligned}
Tr^*_{local,inet} &= \sum_{C_{i,local},C_{j,local}} (T^*_{i,j}S^*_{i,j} + T^*_{j,i}S^*_{j,i}) + \sum_{C_{j,local}} (T^*_{user,j}S^*_{user,j} + T^*_{j,user}S^*_{j,user}) \\
Tr^*_{cloud,inet} &= \sum_{C_{i,cloud},C_{j,cloud}} (T^*_{i,j}S^*_{i,j} + T^*_{j,i}S^*_{j,i}) + \sum_{C_{j,cloud}} (T^*_{user,j}S^*_{user,j} + T^*_{j,user}S^*_{j,user}) \\
costs &= cost_{local,inet}(Tr^*_{local,inet} - Tr_{local,inet}) + cost_{cloud,inet}\,Tr^*_{cloud,inet}
\end{aligned}$$

Integrating applications

Situation

Organizations confronted with many networked applications, but achieving interoperability was painful.

Basic approach

A networked application is one that runs on a server making its services available to remote clients. Simple integration: clients combine requests for (different) applications; send that off; collect responses, and present a coherent result to the user.

Next step

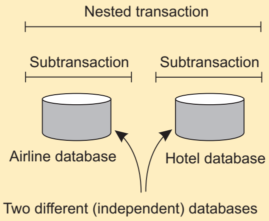
Allow direct application-to-application communication, leading to Enterprise Application Integration.

Example EAI: (nested) transactions

Transaction

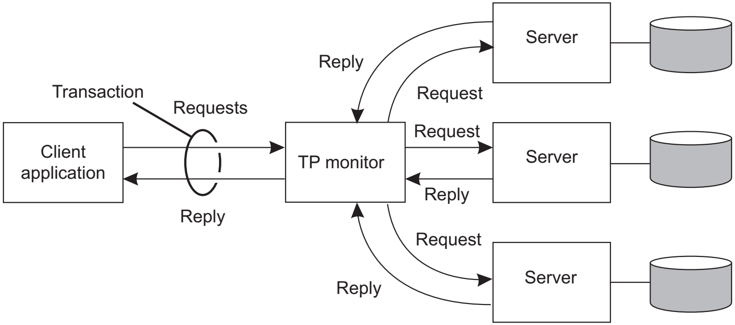
Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Issue: all-or-nothing



- **Atomic**: happens indivisibly (seemingly)
- **Consistent**: does not violate system invariants
- **Isolated**: not mutual interference
- **Durable**: commit means changes are permanent

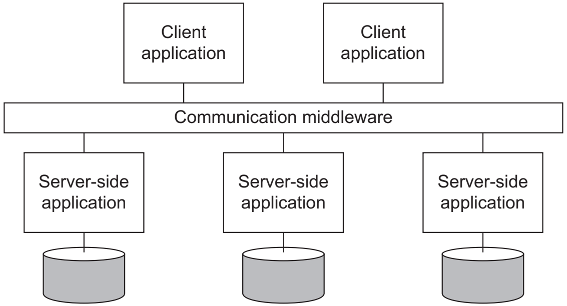
TPM: Transaction Processing Monitor



Observation

In many cases, the data involved in a transaction is distributed across several servers. A TP Monitor is responsible for coordinating the execution of a transaction.

Middleware and EAI



Middleware offers communication facilities for integration

Remote Procedure Call (RPC): Requests are sent through local procedure call, packaged as message, processed, responded through message, and result returned as return from call.

Message Oriented Middleware (MOM): Messages are sent to logical contact point (published), and forwarded to subscribed applications.

How to integrate applications

File transfer: Technically simple, but not flexible:

- Figure out file format and layout
- Figure out file management
- Update propagation, and update notifications.

Shared database: Much more flexible, but still requires common data scheme next to risk of bottleneck.

Remote procedure call: Effective when execution of a series of actions is needed.

Messaging: RPCs require caller and callee to be up and running at the same time. Messaging allows decoupling in time and space.

Distributed pervasive systems

Observation

Emerging next-generation of distributed systems in which nodes are small, mobile, and often embedded in a larger system, characterized by the fact that the system naturally blends into the user's environment.

Three (overlapping) subtypes

- Ubiquitous computing systems: pervasive and continuously present, i.e., there is a continuous interaction between system and user.
- Mobile computing systems: pervasive, but emphasis is on the fact that devices are inherently mobile.
- Sensor (and actuator) networks: pervasive, with emphasis on the actual (collaborative) sensing and actuation of the environment.

Ubiquitous systems

Core elements

1. (**Distribution**) Devices are networked, distributed, and accessible in a transparent manner
2. (**Interaction**) Interaction between users and devices is highly unobtrusive
3. (**Context awareness**) The system is aware of a user's context in order to optimize interaction
4. (**Autonomy**) Devices operate autonomously without human intervention, and are thus highly self-managed
5. (**Intelligence**) The system as a whole can handle a wide range of dynamic actions and interactions

Mobile computing

Distinctive features

- A myriad of different mobile devices (smartphones, tablets, GPS devices, remote controls, active badges.
- Mobile implies that a device's location is expected to change over time ⇒ change of local services, reachability, etc. Keyword: discovery.
- Communication may become more difficult: no stable route, but also perhaps no guaranteed connectivity ⇒ disruption-tolerant networking.

Mobility patterns

Issue

What is the relationship between information dissemination and human mobility? Basic idea: an encounter allows for the exchange of information (pocket-switched networks).

A successful strategy

- Alice's world consists of friends and strangers.
- If Alice wants to get a message to Bob: hand it out to all her friends
- Friend passes message to Bob at first encounter

Observation

This strategy works because (apparently) there are relatively closed communities of friends.

Community detection

Issue

How to detect your community without having global knowledge?

Gradually build your list

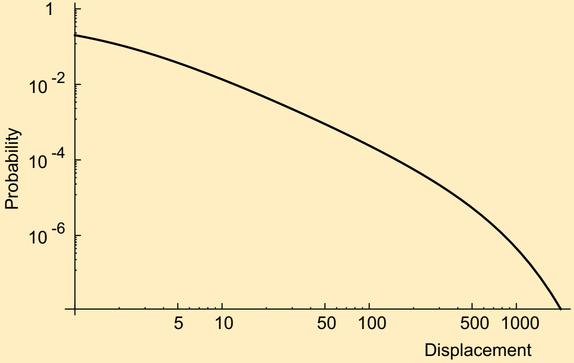
1. Node i maintains familiar set F_i and community set C_i, initially both empty.
2. Node i adds j to C_i when $\frac{|F_j \cap C_i|}{|F_j|} > \lambda$
3. Merge two communities when $|C_i \cap C_j| > \gamma |C_i \cup C_j|$

Experiments show that $\lambda = \gamma = 0.6$ is good.

How mobile are people?

Experimental results

Tracing 100,000 cell-phone users during six months leads to:



Moreover: people tend to return to the same place after 24, 48, or 72 hours ⇒ we're not that mobile.

Sensor networks

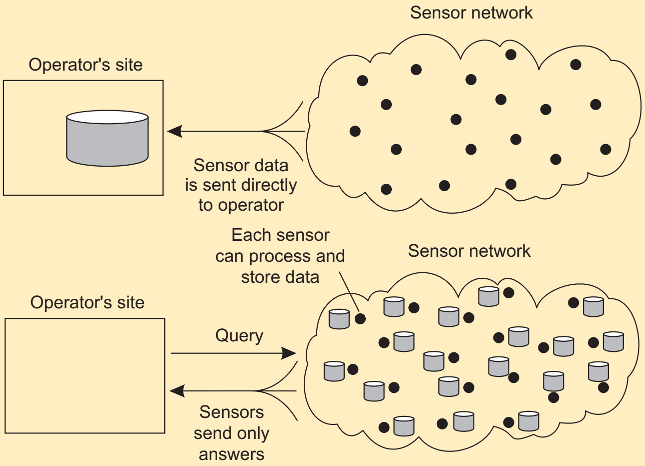
Characteristics

The nodes to which sensors are attached are:

- Many (10s-1000s)
- Simple (small memory/compute/communication capacity)
- Often battery-powered (or even battery-less)

Sensor networks as distributed databases

Two extremes


Sensor network
Operator's site
Sensor data is sent directly to operator
Each sensor can process and store data
Sensor network
Operator's site
Query
Sensors send only answers

Duty-cycled networks

Issue

Many sensor networks need to operate on a strict energy budget: introduce duty cycles

Definition

A node is active during T_{active} time units, and then suspended for $T_{\text{suspended}}$ units, to become active again. Duty cycle τ:

$$\tau = \frac{T_{\text{active}}}{T_{\text{active}} + T_{\text{suspended}}}$$

Typical duty cycles are $10 - 30\%$, but can also be lower than 1%.

Keeping duty-cycled networks in sync

Issue

If duty cycles are low, sensor nodes may not wake up at the same time anymore and become permanently disconnected: they are active during different, nonoverlapping time slots.

Solution

- Each node A adopts a cluster ID C_A, being a number.
- Let a node send a join message during its suspended period.
- When A receives a join message from B and $C_A < C_B$, it sends a join message to its neighbors (in cluster C_A) before joining B.
- When $C_A > C_B$ it sends a join message to B during B's active period.

Note

Once a join message reaches a whole cluster, merging two clusters is very fast. Merging means: re-adjust clocks.