Introduction to Dynamic Distributed Systems
Outline

- Introduction
  - Churn
- Building Applications in Dynamic Distributed Systems
  - Registers
  - Eventual Leader election
- Connectivity in Dynamic Distributed Systems
Dynamic Distributed Systems: Context & Motivations

- Advent of Complex Distributed Systems (p2p, sensor networks, mobile networks etc…)
  - new problems (information dissemination, content distribution and retrieval, service orchestration and composition, location dependent computing, smart spaces etc.)
  - practical solutions on top of a system with variable size in space and time
  - lack of any formal framework
Dynamic Distributed Systems: Assessing assumptions through applications

- Each process autonomously decides to locally run the same distributed application, when joining (it becomes up) and leaving the system.

- It is impossible to know the set of processes participating to the computation because it could be potentially infinite.

- As extremes, in some moments the system could cease its existence as no process is currently active and at some other moment the system is made of tens, or thousands, of active processes.
Dynamic Distributed Systems: Assessing assumptions through applications

- The system
  - does not start with a known and pre-defined setting
  - is just the “sum” of all running entities and their local configurations

- each entity has to learn what the system is at run-time in order to successfully reach system goals
Static distributed Systems

- main characteristics: a predefined setting i.e.,
  - the application knows, directly or indirectly, the set of processes that will participate to the computation.
  - The application also knows if it can exploit synchrony assumptions

- This has a noteworthy consequence: the system can be carefully and "centrally" configured through an appropriate tuning phase in order to get the best performance.

- The application cycle is: Design, deployment, configuration, final deployment, operational

- Air traffic control, Financial systems, Aereospace systems, Egov, Telco service continuity, and many others are examples of static distributed systems
Dynamic system is not a synonymous of scalable system

- Can Applications for static distributed systems be scalable?

  Yes they can!

- As a matter of fact scalability is therefore one aspect of a dynamic distributed systems i.e.,

  “if an algorithms (resp. a system) has to adapt to a dynamic distributed system, the algorithm (resp. System) needs to be scalable, the vice versa is not necessarily true”
Dynamic Distributed Systems: Uncertainty in Distributed Systems

- Static Distributed Systems
  - Lack of temporal knowledge
  - failures
  - unknown communication delays

- Dynamic Distributed Systems
  - Same as in static distributed systems
  - non-monotonic and unknown size of the system (due to churn)
  - neighborhood

- Solid theoretical foundations
- Precise problem specifications
- Rigorously correct solutions
Managed vs. Unmanaged distributed applications (ii)

**Unmanaged Distributed Applications**

- No assumption of a manager or access to equivalent management facilities
- Each process autonomously decides to locally run a component of a distributed application when (a) joining and (b) leaving the system
  - the system and/or its components do not start with a known and pre-defined setting
- “Nice” manageable system model assumptions either cannot be guaranteed or do not last for long
Unmanaged distributed applications: Consequences

- Autonomic/autonomous behavior of entities
- Self-defined, self-instantiating (& self*?) and perpetually evolving distributed system
  - It is impossible to know the set of processes participating to the computation because it changes dynamically and can potentially grow without bounds
  - E.g., the system could cease existing when no process is active, and at other times the system may be made of thousands of active processes

... Dynamic Distributed System
Dynamic Distributed System Model

- Infinite set of processes (nodes)
  - at each time unit the set of processes inside the system is finite
  - unique identifier
  - communication by message exchange
  - Processes do not know each other

- Processes of the system run a computation
  - At each time a subset of processes of the system are part of the computation
  - Processes autonomously decide when enter and leave the computation
    - Churn phenomenon
System vs Computation

Computation Level

System Level
System Churn

Computation Level

System Level
Spectrum of Possible System Models

Orderly

Static Managed Distributed Systems

World

Delay Tolerant Systems

Dynamic Unmanaged Distributed Systems

Chaotic

Platform-based Systems

Cloud Computing

Peer-to-peer

Mobile ad-hoc System

Delay Tolerant Systems
System Architecture

Distributed Computing Abstraction

Connectivity Layer

Today

Next Seminar
Modeling Churn
Churn Models

- Infinite Arrival Model
- Train Model and Crowd Model
- Deterministic Function based Model
Infinite Arrival Model

- In each infinite run the number of participant is unknown
  - **Concurrency Level**: maximum number of processes that may participate to the algorithm at the same time
    - Finite
    - Bounded
    - unbounded
  - **Required Participation**: defines the minimum and maximum number of participant required from a given algorithm
Train Model and Crowd Model

**Train Model**
- Join and leave of processes are periodic
- The number of processes that join is equal to the number of processes that leave
- Such number is the same for the entire computation

**Crowd Model**
- Generalization of the train model
- Processes still join and leave the computation in groups but...
  - The join and leave occur at and arbitrary time
  - The number of changes in no more fixed

C processes join and C processes Leave

C processes join and C processes Leave

C’ processes join and C’ processes Leave
Deterministic function based Model (1/2)

- **Join Distribution** $\lambda(t)$
  - Is a discrete time function that returns the number of processes that invokes the join operation at time $t$

- **Leave Distribution** $\mu(t)$
  - Is a discrete time function that returns the number of processes that have left the computation at time $t$

---

**OBS.** By using $\lambda(t)$ and $\mu(t)$ it is possible to characterize the network size variation
Deterministic function based Model (2/2)

- **Node Distribution N(t)**
  - Let $n_0$ be the number of processes in the computation at the initial time $t_0$. $N(t)$ is the number of processes within the computation at any time $t > t_0$.
    - Recursive definition $\Rightarrow N(t) = N(t-1) + \lambda(t) - \mu(t)$ with $N(t_0) = n_0$

Infinite arrival model

*Churn rate. The* number of processes remains constant (equal to $n$), in every time unit $c \times n$ processes leave the system and the same number of processes join the system.

- General churn model has been investigated in [*Baldoni – Bonomi – Raynal SIROCCO2009*]
Register
Register: definition

A register is a shared variable accessed by processes through *read* and *write* operations.
Distributed Systems: register abstraction

- **Multiprocessor machine:**
  - Processes typically communicate through registers at hardware level

- **Distributed message passing system:**
  - No physical shared memory
  - Processes communicate exchanging msg over a network

- The set of these registers constitute the physical memory

- Register abstraction support the design of distributed solution, by hiding the complexity of the underlying message passing system and the distribution of the data
Register operations

A process accesses a register through:

- **Read operation**, \( \text{read}() \rightarrow v \): it returns the “current” value \( v \) of the register; this operation does not modify the content of the register;

- **Write operation**, \( \text{write}(v) \): it writes the value \( v \) in the register and returns \( \text{true} \) at the end of the operation

Each operation starts with an invocation and terminates when the corresponding response is received
Register: Assumption

- A register stores only positive integers and it is initialize to 0

- Each value written is univocally identified

- Processes are sequential: a process cannot invoke a new operation before the one it previously invoked (if any) returned
Register: Notation

\((X,Y)\) denotes a register where \(X\) processes can write and \(Y\) processes can read

- \((1,1)\) denotes a register where only a process can write and only a process can read. It is a priori known which process can write and which can read

- \((1,N)\) denotes a register where a single process, a priori known, can write, and \(N\) processes can read
Process Behavior

- When a process wants to participate in the computation, it executes the `join()` operation:
  - It is a not instantaneous operation.
  - A process participates effectively in the computation only after it has completed the `join()` operation.

- When a process no longer wants to participate in the computation, it leaves:
  - Implicit leave: processes stop to execute the regular register protocols.

- Processes do not crash.

- **Def.** A process is **active** from the time it returns from the `join()` operation until the time it leaves the system:
  - $A(t)$ denotes the set of active processes at time $t$.
  - $A([t_1, t_2])$ denotes the set of active processes for the whole interval $[t_1, t_2]$.

- At any time $t$, a subset of processes (active processes) belonging to the distributed system participates in the computation.
Regular register computation

- A protocol implementing a Regular Register in a Dynamic Distributed System has to satisfy the following properties:
  - **Liveness**: If a process invokes a read or a write operation and does not leave the system, it eventually returns from that operation.
  - **Safety**: A read operation returns the last value written before the read invocation, or a value written by a write operation concurrent with it.

- Each process participating to the register computation maintains a local copy of the register and an associated timestamp.
Impossibility

- **Theorem**
  - *It is not possible to implement a Regular Register in a distributed system if the churn is continuous*  

- **Proof Intuition**
  - It is not possible to know when the operation is terminated because processes continuously change  

---

To Implement a Register the churn has to be quiescent
Eventually Synchronous System

- Two communication primitives are available:
  - **Point-to-point primitive**
    - There is a time $t$ and a bound $\delta$ such that any message sent at time $t' > t$ is received by time $t' + \delta$ from any process that in the system for all the period $[t', t' + \delta]$
  - **Broadcast primitive**
    - There is a time $t$ and a bound $\delta$ such that any message broadcast at time $t' > t$ is delivered by time $t' + \delta$ from any process that in the system for all the period $[t', t' + \delta]$
Regular Register in Dynamic Distributed Systems: write()

The writer process $p_w$ wants to write the value $v$. $p_w$ sends a broadcast message $(\text{WRITE}, v, sn)$ and waits for replies. A subset of processes participate to the register computation. Only processes that belong to the computation when $p_w$ starts the write and that remain in the computation throughout the whole time of the write will maintain the updated copy of the register.

In the meanwhile, processes join and leave the computation. Active Processes maintains the state of the computation.
Joining the Regular Register Computation

Assumption: There is always a Majority of active processes

1. A joining process broadcast an inquiry message to obtain the value of the register
2. It waits for at least a majority of replies
3. As soon as it has received the replies, it select the most updated value
4. It becomes active
   a) It replies to pending join
Vertical Quorums for Register Validity in Asynchronous Periods

Termination. Let us assume that $|A(t)| > n/2$ (i.e., majority of processes is active at any time), if a process invokes join(), read() or write(), and does not leave the system, it terminates its operation.

Safety. Let us assume that $|A(t)| > n/2$, a read operation returns the last value written before the read invocation, or a value written by a write operation concurrent with i

Validity of the read
During asynchrony periods to be sure to read the last written value you need to read/write registers from a majority of processes in the system (you do not have anymore the guarantee that messages are delivered within a known bound)