Distributed and Real-Time
Cyber-Physical Systems

Davide Galassi <nackday@datawok.net>
Based of work of Prof. Andrea Bondavalli <andrea.bondavalli@unifi.it>

Table of Contents

Foundations

Computing Evolution

Industry 4.0

1. First Industrial Revolution (1800): mechanization, water and steam power
2. Second Industrial Revolution (1900): mass production, assembly line, electrical energy
3. Third Industrial Revolution (1970) : computer automation, electronics and IT
4. Fourth Industrial Revolution (today) : cyber-physical systems

Mainframe computing (60's-70's): large computers to execute data processing applications.
Desktop computing (80's-90's): one computer at every desk to do business and personal activities.
Ubiquitous computing (00's): numerous computing devices in every place. Invisible part of the environment. Millions for desktops vs billions for embedded processors.
Cyber-Physical Systems (10's): systems able to interact with the surrounding physical world.

Example of applications of CPS are environmental control, electricity and natural gas distribution, physical infrastructure monitoring and control (e.g. railway systems).

Monolithic Systems

Starting from **mainframes**, computers were usually characterized by distinguishable services that are not clearly separated in the implementation. Rather than containing separate components they are interwoven into a singular, monolithic system.

A System of Systems stems from the modular integration of existing legacy systems and new systems that have been designed to take advantage of this integration. The components are normally operated by different organizations (example: smart metering infrastructure includes legacy smart meters).

Characteristic	**Monolithic**	**System of Systems**
Scope	Known	Unknown
Clock Synchronization	Internal	External
Structure	Hierarchical	Networked
Requirements and Specs	Fixed	Changing

Evolution	Version Control	Uncoordinated or Managed
Testing	Test Phases	Continuous
Implementation	Technology Given and Fixed	Unknown
Faults	Exceptional	Normal
Control	Central	Autonomous
Emergence	Insignificant	Important
Development	Process Model	Independent for each Component

SoS Viewpoints

Most of the cost of SoS development is in the engineering phase, the hardware requirements relatively cheap. **Reduction of cognitive complexity** of large systems has thus great economic significance, reducing the probability of design errors.

Cognitive-Complexity: how much mental effort is required to understand a given scenario for the given purpose by the average user of a group. Understand the behavior of a system means that a **mental model** that establishes **causal links** between: observable inputs, state, observable outputs of a system has been formed.

A SoS can be difficult to be analyzed as a whole, thus the main characteristics are summarized as **viewpoints**:

- **Fundamental concepts**: definition of SoS and related parts.
- **Time**: time management and its role.
- **Data and State**: information exchanged between the parts. Representation and Metadata.
- **Actions and Behaviour**: the dynamics. Event-based view or state-based viewpoint.
- **Communications**: protocols and role.
- **Interfaces**: the means that allows the components to interact.
- **Dynamicity and Evolution**: short term (dyn.) and long term (evol.) changes.
- **Design and Tools**: architecture and dynamic behavior design methodologies
- **Dependability and Security :** confidentiality, integrity, safety, availability and reliability
- **Emergence :** novel behaviors that manifests only when considering the system as a whole.

Core Concepts

Interval of Discourse (IoD): time interval of interest.
Universe of Discourse (UoD): set of entities and relations among the entities that are of interest when modeling the selected view during the IoD.
We must identify the objects that have distinct and self-contained existence in the UoD during the IoD.

Entities: objects that have a distinct and self contained existence:

- **Thing**: a physical entity, adhering to physical time and belongs to physical space.
- **Construct**: product of human mind, adhering to a discrete time-base, belongs to cyber space.

System: entity capable to *interact* with the environment and *sensitive* to progress of time. The same system can react differently depending on the environment or time.

Environment: entities that are not modeled as part of a system but can interact with the systems.

System boundary: dividing line between a system and environment or another system. In SoS this boundary is not well defined and may change frequently over time. CS comes and go from the system.

System Architecture: *blueprint* of a design that establishes the overall structure, the major building blocks and the interaction between these major blocks and environment.

Autonomous System: a system that tries to achieve its objectives, and provide its services, without guidance by other system.

Constituent System (CS): an autonomous system that is part (subsystem) of a SoS.
A CS is itself a system, and thus can itself be dissected into sub-CSs. The process is recursive and terminates when the internal structure of a subsystem is of no further interest (atomic components).

Cyber-Physical System (CPS): a system consisting of a computer system (cyber), a controlled object (physical) and possibly interacting humans.

- **Cyber**: computation, communication, and control are discrete, logical and switched.
- **Physical**: systems governed by the laws of physics and operating in continuous time.

Interacting human can be:

- *prime mover*: interacts with the system according to his/her own goal.
- *role player*: acts according to a given script and shall be stimulated.

CPS Entourage: parts of the system that are external to the cyber part, i.e. physical parts and interacting humans.

System of System (SoS): integration of a finite number of CS which are independent and that are networked together for a period of time to achieve a certain higher goal. System boundaries are thus defined only for a period of time and, potentially, the SoS may exist only during the IoD.

SoS classification:

- **Directed**: central managed purpose and ownership for all CSs (e.g. micro kernel).
- **Acknowledged**: independent ownership of CSs but aligned purpose (e.g. smart grid).
- **Collaborative**: independent ownership of CSs and individual purposes (e.g. a horde).
- **Virtual**: lack of central purpose and alignment (e.g. cars traffic).

Data and State

Data: artefact created for a specific purpose. In cyber-space it is a bit-string needing an explanation to arrive at its meaning (the information).

Information: a proposition about a state or an action in the world, is composed by two data elements:

- **object-data**: the payload.
- **meta-data**: payload explanation.

State: the totality of information from the past, at a given instant, that can have an influence on the future behavior of a system. State concept is meaningless without the concept of time.

State Space: totality of all possible values of the state variables.

Actions and Behavior

We can observe the dynamics of the system that consist of discrete variables.

Event-based view: observe values of every state variables at the beginning, record all events and timestamps in a trace. The amount of data generated by the event-based view **cannot be bounded**.

State-based periodic view (Sampling): observe the relevant state variables at sampling points. Granularity of sampling is important, we can loose configurations but data is bounded.

Execution Time: time required to execute an action on a system.

An **action** can be: **time triggered** and **event triggered**.

Function: mapping of input data to output data.

Behavior: timed sequence of effects of input and output action that **can be observed** at an interface of a system. Behavior is **deterministic** if given a set of inputs at a defined instant the future outputs can be predicted. **Service**: the intended behavior of a system that can be exploited by an user.

Communication

A SoS cannot exist without transfer of information between its CSs.

Communication happens with transport of messages from a sender to one or more receivers. Transport should have high **dependability** and short **duration**.

Communication protocol: set of rules that govern a communication action.

Message: data structure used to exchange information.

- **Datagram**: a best effort message (sporadic)
- **PAR**: Positive Acknowledgment or Retransmission (sporadic)
- **TT**: Time Triggered, error controlled transport service for transmission (periodic)

Stigmergy: information exchange via changes in the environment. The coordination is not direct, the trace left in the environment by an action stimulates the performance of the next action by the same or different agent.

Environment Dynamics: autonomous environment processes, not explicitly modeled as CS, that cause the change of state variables in physical environment.

Interfaces

Points of interaction of systems with each other and environment over time.

Interaction: exchange of information at connected interfaces

Channel: a logical or physical link that transports information among systems through their connected interfaces.

Relied Upon Interface (RUI): interface of a CS where the services are offered to other CSs. Are the interfaces used to form the SoS.

- **RUPI**: physical interface for things/energy exchange.
- **RUMI**: cyber interface for message exchange.
- **RUS**: cyber interface for service exchange.

Time Synch Interface (TSI): enables external time synchronization to establish a global time-base.

Utility Interfaces : configuration (C-Interface), diagnose (D-Interface), monitor and update the system

Dynamicity and Evolution

Dynamicity (short term changes): capability to react promptly to changes in the environment.
Example: accidental blockchain fork.
Reconfigurability: capability to adapt internal components to mitigate internal failures or improve service quality.

Evolution (long term changes): process of gradual, progressive change. Primarily resulting from environment changes. In SoS context means maintaining and optimizing the system.

- **Managed**: process of modifying the SoS to keep it relevant. Usually the process is under the control of an Authority to avoid unwanted emergent behaviors. Example: blockchain hard fork.
- **Unmanaged**: modification occurring as a result of and independent changes in some of its CSs. Example: faster internet node.

Authority: association with the right to demand for managed changes in order to keep the SoS relevant to stakeholders. Authority establish RUI specifications and how changes for evolution are rolled out.

Design and Tools

From conceptual thoughts to requirements and architecture design.
Design: process of defining an **architecture**, **components**, and **interfaces** that satisfy the requirements.
Modularity: technique to build large systems integrating simpler and reusable modules.
A good design is needed to have a component that it is:

- **Evolvable**: modifiable to be used in new contexts.
- **Flexible**: can be adapted to future contexts.
- **Robust**: performs well under different environments.
- **Testable**: can be easily tested.

Dependability and Security

Threat: circumstance or event that potentially impact an organization operations, assets, individuals via unauthorized access, destruction, disclosure, modification of information or denial of service.
Vulnerability: weakness in a system design or implementation that could be exploited and that may originate a threat.
Risk: a measure of the extent to which an organization is threatened. The likelihood and the impact.

Threats

- **Fault**: can be an internal defect in the system or an external fault (cause by external failure).
- **Error**: a fault that is activated, can propagate through components.
- **Failure**: an error that reach the system boundary, i.e. the service interface.

Security Attributes

- **Confidentiality**: absence of unauthorized disclosure of information.
- **Integrity**: absence of improper system state alterations.
- **Authenticity**: the guarantee of the identity of the information creator.
- **Non repudiation**: the information creator cannot deny its actions.

Dependability Attributes

- **Availability**: readiness for service.
- **Reliability**: continuity of service.
- **Safety**: absence of catastrophic consequences.

Robustness: dependability with respect to external faults.

Means to improve dependability with respect to faults:

- Forecasting : estimate the number, the future incidence and consequences.
- Prevention: prevent occurrence or introduction of faults (e.g. unit testing).
- Removal: means to remove faults after introduction.
- Tolerance : avoid service failure in presence of faults, restore stable system state.

Emergence

A phenomenon of a whole at the macro-level is emergent if and only is is of a new kind with respect to the non-relational phenomena of any of its proper parts at the micro level.

An Emergent phenomena can be: either expected or unexpected, beneficial or detrimental.

Interfaces

Advances in telecommunications and automation allowed integration of previously isolated systems .
Central to the integration of CPS are their interfaces, points of interaction of CSs within each other and the environment.

Key challenges:

- **Identification** of proper interfaces.
- Proper **specification** and **standardization** of interfaces.
- Managed **modification** of interface specifications.

Time plays an important role, especially for appearance/avoidance of emergent behaviors.
Many attributes of interest in SoS, we focus on behavioral attributes, i.e. interactions among architectural elements of SoS.

Architectural Elements:

- Itom (object data + meta data) : unit of interaction.
- Component System: Itom processing entity that exchanges Itoms with its environment.
- Environment: all entities a CS can interact with.

Itom : “timed proposition about some state or behavior in the world”.
Can be sent as object data of another Itom (recursive structure), avoids misinterpretation of information, enable information context independence.

Constituent System: is a CPS composed of **cyber** part (digital machine adhering to discrete progression of sparse digital time), a **physical** part interacting with environment via sensors/actuators and eventually humans (adhering to the progression of dense physical time).
Proper integration of physical and digital time is essential.

Interface specification: defines the CS’s interface capabilities, the purpose is realization of CPSoS emergent services.

Computer System of CS processes Itoms according to the Interface specification, exact details of implementation are irrelevant/hidden.
In physical space Itoms are sent via **actuators** and received via **sensors** to and from the CS entourage.
In cyber-space Itoms are sent and received as (timestamped) **messages**.

Environment: entities and actions that are not part of the system but have the capability to interact with the system. Distance is fundamental for interactions (force fields).

Cyber Environment is a distributed Itom processing system (e.g. IP based Internet). Message based communication via both direct and indirect channels (e.g. shared memory over cyber space). Shared memory is subject to **cyber-dynamics**, time sensitive autonomous processes acting on cyber data.

Physical Environment consists of things and physical fields. Properties can be modeled as a dynamic network of **physical state variables** described by environmental model**.** Subject to **environmental dynamics**: time sensitive autonomous processes acting on physical state variables. Actuators write state variables, sensors read state variables.

CS Entourage: all the physical things interacting with the CPS, allows stigmergic information flow.

Layers

Enable discussion of interface properties and their definition in interface specificatios.

Cyber-physical interface layer

Level of messages and things/energy. Interactions realized by concrete tech and sensors/actuators. Too much detailed (low level) to study SoS properties (e.g. emergence, evolution, dynamicity). At lowest level even cyber interactions are realized by physical interactions. Interface details in the **CP-Spec** composed by **M-Spec** and **P-Spec** (properties of sensors/actuators e.g. sample rate).

Physical Interface

Itoms represented by a bit-pattern, explanation deduced by placement of sensors/actuators and design.
P-Spec and environmental model allows **simulation** of stigmergic channels.
After sampled raw-data has been refined into an Itom, the raw data becomes irrelevant.

Cyber Interfaces

Produce and consume messages according to M-Spec, consisting of: transport specification (properties of messages at transport level), syntactic specification, semantic specification. Can be structured as **ports** (channel endpoints) where messages are placed and fetched.

Informational interface layer

Level of Itoms. Abstracts the low level information origin and technology (cyber or physical irrelevant). **I-Spec**: Itom types, temporal, security and dependability properties.
E.g. emergency breaking of a car at CP level can be realized by a stigmergic channel (break light in front and human operator behind) or by M2M cyber channel. But the Itom at information level is the same.

Communication can be **direct** or **indirect.** In indirect communication environmental and cyber dynamics should be considered. If there is an indirect communication, interacting channel can be modeled by an **Environmental CS (ECS)** that apply environment-dynamics to the Itoms flow. A system which effect is considered environmental dynamics can not be modeled explicitly.

The informational interface is the most suitable layer to study emergence, dynamicity and evolution.

Frame-based Sync Dataflow Model (FSDM)

Interface specification does not describe the behavior or CS environment itself. Is needed an appropriate execution semantic. **FSDM** is employed to model from design to analysis dependable distributed RT systems. Offers a semantics for study behavioral properties of CPSoS at **informational layer**.
Dense time segmented in **frames** (periods) consisting of:

- **Synchronization**: write output and read input to/from environment.
- **Processing**: calculate next state and output from input and current state

Frame duration is short enough such that system appropriately reacts to changes in the environment (e.g. keyboard 50 ms, car crash sensor 1ms).

Service interface layer

System behavior structured as **capabilities**, by eventually grouping sub-Itom channels. Abstracts individual information channels. E.g. database service is composed of request/response Itom channels.

S-Spec: set of quality metrics to allow independent observer to determine quality of provided service. Providers offers capabilities under a **Service Level Agreement** (SLA).

Service-oriented architecture (SoA)
Building blocks: providers, consumers and brokers.
Principles:

- registry: repository of service specifications (S-Specs);
- discovery: service consumers match their requirements with the registry;
- composition: multiple services are integrated into a new, higher level, service;
- abstraction: implementation of a service are unimportant (black box);
- stateless: providers don’t maintain a context for each consumer (easier switch)

Different from modularity: SoA is more about how to compose an application by integrating distributed, separately-maintained and deployed software components.

Implementation: SoA can be implemented with Web services to allow uniform access over ubiquitous Internet protocols that are independent of the platform. Examples: SOAP, CORBA, REST. Others view the realization of SoA in SaaS, PaaS and cloud computing in general.

Criticism: SoA has been conflated with Web-services, however, Web services are only one option to implement the patterns that comprise the SOA style. In the absence of native or binary forms of remote procedure call (RPC), applications could run more slowly and require more processing power, increasing costs. Most implementations do incur these overheads, but SOA can be implemented using technologies that do not depend on remote procedure calls or translation through XML.
Further services may belong to different (even competing) organizations creating a huge trust issue.

Relied Upon Interface

RUI : CSs interfaces used to provide emergent CPSoS services.
Time-Sync Interface (TSI) : realize sparse global timebase.
Utility Interfaces : C-Interface (configuration) and D-Interface (diagnostic).

Short-term changes (dynamicity) need to be considered in RUI specification, long-term changes (evolution) affect how RUI specifications are updated.

RUI at Cyber-Physical layer

RUMI: direct or indirect (e.g. may be subject to cyber/environment-dynamics)
RUPI: overlapping entourage allows stigmergic information flow. All interactions are indirect.

RUI at Informational Layer

Abstracts the underlying type (RUMI/RUPI) and thus the concrete implementation technology.
This layer simplifies the global system view abstracting from the lower layers technology.
Context sensitivity by using explicitly defined Itoms (data and metadata).
Focus on direct and indirect information flow among CS. Indirect channels modeled by the instantiation of an additional **Environment CS** (ECS).
If the implementation shows a different emergent behavior with respect to the model, then then there are possible hidden channels in the lower layer. A **Hidden channel** is a latent information flow among CSs not considered in the model.

RUI at Service Layer

RUS: relied upon service. Described in the S-Spec.
RUS is a behavioral abstraction over one or more unidirectional Itom channels (group together Itom channels) and specifies interaction pattern, i.e. the sequence of operation related Itoms.
An emergent CPSoS service is modeled as set of dependencies of required RUSs. Required RUSs must be provided by a CS that wants to benefit from emergent CPSoS service. CSs doesn't need to provide RUS directly, can use composition of the RUSs provided by other CSs.
SLA: Service Level Agreement

RUI Evolution Handling

Design modifications triggered by changes in environment: e.g. advances in tech or business needs.
Increase CPSoS business value, retro compatibility and prevent obsolescence.

Unmanaged evolution: no guidance and central purpose. CS owners freely change their CSs.
Managed evolution: orchestrated by an Authority to supervise the operation. Appropriate for directedm acknowledged and collaborative SoS.

The **Authority** may introduce changes in the RUI while maintaining retrocompatibility. Has the capability to incentivate the adoption of a changed RUI specification and to select appropriate standards. Is usually composed by key CPSoS stakeholders (e.g. CS manufacturers and governments).
The authority can authorize and publish RUI specifications.

A **minor evolution** step affects only the cyber-physical layer, a **major evolution** step affects the informational and service layer (more likely to introduce detrimental emergence).

If emergence manifests in a minor evolution step then there were a latent hidden channels that we've not considered in the higher level layers.

In **time-aware** CPSoS the global timebase can be used to temporally coordinate the execution of evolutionary steps. Can be useful a dormant period before restart the CSs to be more confident about the alignment.

Emergence

“The Whole is Greater than the Sum of its Parts” *Aristotele*

Emergent phenomena: a phenomena of a **whole** at the macro-level is emergent if is of a new kind with respect to the **parts** at the micro level.

While we design our SoS we want to predict emergent behaviors and manage their appearance.

Emergence attributes

- **Domain**: structure, behavior, properties.
- **Predictability**: unpredictable, stocastic, deterministic.
- **Explanation**: none, in principle, by simulation rules, by analytic rules.
- **Consequences**: positive, none, negative.

Multi-level hierarchy

Understand huge amount of information requires an appropriate **modeling structure**. One modeling technique is a recursive structure called **multi-level hierarchy**. At the level of interest an entity, the whole can be dissected into a set of sub-systems, the parts. Focusing on the lower level, each of these parts can be viewed as a whole itself. Lowest layer constituent is called a **component** or elementary part.

Each level possesses its unique laws, the phenomenon of emergence is always associated with levels of a multi-level hierarchy.

Holon: a two faced character of an entity. Considered a whole at the macro level and an ensemble of parts at the micro level. (Greek origin “*holos*” means *all* and “*on*” means *part*). At the macro level the holon is accessed via interface while at micro level the parts are confined and interacting.

Emergent behavior is mainly associated with control hierarchies and causal loops.

Control hierarchy. The macro-level must on one side constrain the freedom of the behavior of the parts but on the other side abstract from it, thus allowing some degrees of freedom (no control means chaos, absolute control means determinism).

Sources of control:

- **authority from outside**: at the same level, an outer dedicated entity controls the components.
- **authority from inside**: the higher level is equipped with causal powers so that it can inflict effects on the lower level that is causing it (causal loop).

Causation.

- **Upward-Causation**: interaction of parts at micro level cause because of the laws of physics or imposed rules, the whole at macro-level.
- **Downward-Causation**: the ensemble of the parts (the whole) constrains the behavior of the parts at the micro-level resulting in a **causal loop**.

A very likely causal loop source can be searched among hidden stigmergic channels.

Conjecture: in a multi-level hierarchy emergent phenomena can only appear if there is a causal loop. Linear cause-effect relations cannot explain emergent phenomena.

Supervenience. Relation between the emergent phenomena of adjacent levels in a hierarchy.

- **Sup-1** : a difference in the parts may give the same emerging phenomena.
- **Sup-2** : a difference in the emerging phenomena requires a difference in the parts.
  Macro-level-diff → Micro-level-diff (the inverse doesn't hold)

Because of Sup-1, we can abstract from many of the components details greatly simplifying the modeling and engineering of the system.

Example of Sup-1: A transistor is internally unique but its holon is equal to the other transistors.

Types of interactions:

- **physical**: *synchronic* interactions characterized by distance, frequency and force fields. Moving up in abstraction hierarchy the distance increases thus force fields and frequency decrease.
- **informational**: *diacronic* exchange of Itoms across messages or stigmergic channels. Emergent behavior are caused by this type of interactions.

Emergence Explanation

At the macro level new laws may be introduced to be able to describe the phenomena appropriately.

Inter-ordinal laws (**bridge-laws**) to relate established laws at micro-level with the new concepts.

Proper conceptualization of emergent phenomena can lead to an abrupt simplification at the next higher level. For example, the classical mechanics laws can be said to emerge as a limiting case from the quantum mechanics laws applied to large enough masses. Quantum mechanics is generally thought as more complicated than classical mechanics but new rules that simplify are introduced.

Some philosophers take as emergent only phenomena that cannot be explained (not reducible to) the state of knowledge about properties and laws that govern the parts at the micro-level. But what constitutes an acceptable explanation? And what is the reference for the state of knowledge (can be subjective and changes over time). Thus, from this perspective a phenomena can be emergent only for some persons or in some moments in time.

Scientific explanation (Reductionism). Given *statements of the antecedent condition* and *general laws* then a ***logical deduction*** *of the description* of the empirical phenomenon to be explained *is entailed***.**

General laws are universally valid while rules are structure dependent and local. Special case are imposed rules (e.g. a game).

Examples of explained Emergence.
Deadlock: novel phenomena is "*permanent halt*". Downward causation is realized via indirect information transfer via the semaphore variables, is not predictable because of indeterminism in task execution. Best programming practices to avoid it in the single programs (components).
Fault Tolerant Clock: novel phenomena is "*tolerance of clock failures*". Downward causation: the time average of the ensemble of clocks inflicts a state correction to a local clock. The frequency of a physical oscillator cannot be changed (upward causation).
Game of Life: Glinder movement along the diagonal is an emergent phenomena.

Awareness. With proper **observation** and **documentation** of interactions between the CS the occurrence of emergent phenomena in SoS can be predicted and (hopefully) explained. Sometimes is difficult, or impossible, to have complete awareness during SoS design (e.g. system boundaries are not fixed). Unexpected emergence is caused by ignorance about: behaviors of each CS, possible side channels among CS, impact of the environment.

Issues

CPSoS *Design model* may not take into account emergent effects that cause a deviation from the intended behavior. Emergence is **diachronic** (develops over time) thus a continuous observation of the system behavior is important in safety-critical SoS systems. Detect the start of anomaly e.g. via pattern matching.

Legacy CS may come with unknown development faults, bugs, vulnerabilities. Their specifications may be incomplete or incorrect. Integration in a SoS may cause unexpected emergence.

Emergent phenomena are caused by interactions among CS that close a causal loop such that the whole affects the behavior of an individual part at the micro-level. (e.g. the fault tolerant clock is composed by the CS clocks that exchange info using messages, the FT clock influence the single clocks to synchronize them).

To detect actions that lead to emergence expose all information flow channels, search for causal loops, identify capacity limits, look for patterns. When an unexpected emergent behavior is detected, is important to find a proper rational explanation and identify the side channels.
Modern detrimental example: **Meltdown** Intel exploit.

Time and Clocks

Systems are, by definition, sensible to the progression of time, thus changes in a SoS depends on time. In SoS a **global** notion of time is required in order to synchronize the CS to achieve the objectives (e.g. conflict-free resource allocation, security protocols, real time data limits, etc.)

Time. A continuous measurable *physical quantity* in which events occur in a sequence proceeding from the past to the future.

Timeline. A ***dense*** directed line denoting the progression of time from past to future.

- **Instant**: cut on the timeline (totally ordered).
- **Event**: a happening at an instant (partially ordered).

Interval. Segment on the timeline between two instants.

Events order:

- **Temporal**: depends on the happening instant.
- **Causal**: depends on the cause-effect relationship among events.

Events are only **partially ordered**, since simultaneous events are not in the order relation. Can be totally ordered by introducing a criterion to order simultaneous events (e.g. id of the node where the event occurred).

Cycle: sequence of events that once arrived at a final state restarts from the initial state.
Period: a cycle with constant duration between start states.

Clocks

A clock is a device that contains a counter and increments this counter periodically.

Digital Clock: autonomous system that consists of an **oscillator** and a **register.** When the oscillator completes a period an event (**tick**) is generated that increments the register value.
Reference Clock: hypothetical clock with granularity smaller than any duration of interest.
Coordinated Clock: synchronized within stated limits to a reference clock that is spatially separated.

Clock granularity: reference clock ticks between any two consecutive ticks of the clock.
For example: if between two clock ticks there are 100 reference clock ticks and the reference clock has a granularity of 1 ms, then the clock has a granularity of 100 ms.

Nominal frequency. The desired frequency of an oscillator.
Individual clock frequency can deviate from **nominal** frequency.

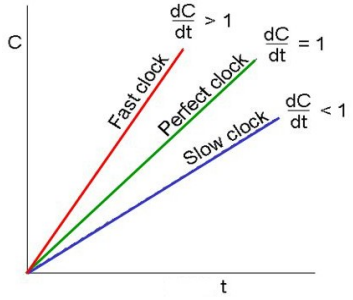
Frequency Drift. Phenomena where clock does not run at exactly the same rate as a reference clock.
Given Hz_r and Hz_c the frequencies of the reference clock and the clock *c*, respectively:

$$\textbf{\textit{Drift}} = Hz_c / Hz_r$$

If the clock *c* has highest frequency than the reference the drift is > 1.

$$\textbf{\textit{Drift rate}} = |\, Drift - 1 \,|$$

Real clocks have drift rates from 10^{-2} (poor quality) to 10^{-8} (high precision quartz clocks).



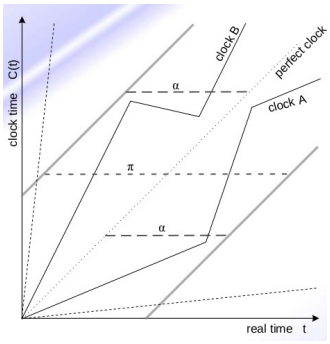
Drift is due to aging, changes in the environment and other factors external to the oscillator.

Wander. Long-term phase variations of the significant instants of a timing signal from their ideal position on the time line (variations frequency less than 10 Hz).
Jitter. Short-term phase variations of the significant instants of a timing signal from their ideal position on the time line (variations frequency greater than or equal to 10 Hz)

The offset of clock *c* with respect to the reference clock *r* at tick *i* is called the **accuracy**. The maximum offset over all ticks is called the **accuracy of the clock**.
The difference between the times of two different clocks is called **clock skew** (offset).
Given an ensemble of clocks the maximum skew between any two clocks of the ensemble is called the **precision** of the ensemble of clocks.

The process of mutual synchronization of an ensemble of clocks in order to maintain a bounded precision is called **internal synchronization**.
The process of synchronization with the reference clock is called **external synchronization**.

If all clocks of an ensemble are externally synchronized with an accuracy *A*, then they are internally synchronized precision is at most *2A*. The opposite is not true.



α = accuracy
π = precision

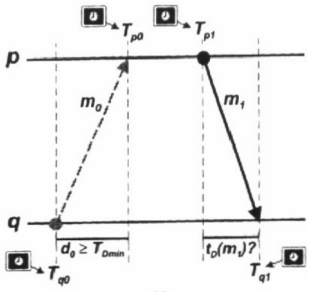
Accuracy is paramount for clock alignment with the reference clock. A high precision clock ensemble can drift together from the reference clock.

Given an ensemble of clocks with a drift rate of δ, to maintain a precision of at least π we shall synchronize the ensemble clocks with the reference clock at least every $\pi / 2\delta$.

Unsynchronized local physical clocks

Local time references and timestamps can only related if they originate from the same clock.
Assuming local clocks have a bounded drift rate, using **time chains** distributed durations can be approximated without explicit existence of global clocks.

For example, we want to evaluate the transmission delay of a message m_1, $D(m_1)$



Process *q* knows $\Delta_q = T_{q1} - T_{q0}$ (Round Trip Time)
Process *p* knows $\Delta_p = T_{p1} - T_{p0}$ (Execution Delay)
Process *p* sends Δ_p with m_1
We know that $d_0 \geq D(min)$, the minimum message delay
So $D(min) \leq D(m_1) \leq \Delta_q - \Delta_p - D(min)$
with $D(min) \leq (\Delta_q - \Delta_p)/2$
The **error** is between *0* and $G = D(max) - D(min)$

The idea is used by both the Christian's algorithm and NTP for clock synchronization.

Global Time and Standards

In a distributed system, local clocks are synchronized in order to generate a common notion of time, a **global time** in the distributed system. Such global time is an **abstract notion** that can only be approximated by the clocks in the nodes.

A **dependable** global time is required to measure duration between events that happens in systems with autonomous oscillators. For example to measure how much time a message takes to go from one node to the other.
A SoS global time enables the interpretation of timestamps in different CS in order to:

- Limit validity of RT control data
- Synchronization of IO actions across nodes
- Specify temporal properties of interfaces
- Perform prompt error detection
- Strengthen security protocols
- Conflict free resource allocation
- Reach consensus

UTC time standard is aligned to day duration (one day is 86400 seconds). Since the earth rotation is not so exact, a leap second should be introduced approximately every 3 years.
TAI time is similar to UTC but is instead a *chronoscopic* timescale, i.e. monotonic with no introduced leap seconds. The 1st January 1958 TAI time was started as equal to UTC.

Reasonableness Condition

Given:

G the global time granularity (macrotick granularity).
Π the precision of the clock ensemble

The global time t is called reasonable, if all local implementations of global time satisfy the **reasonableness condition**: $G > \Pi$.
The condition ensures that the synchronization error is bounded to less than the duration between two macroticks. If the condition is respected by all network nodes, the timestamps of an event detected in parallel by multiple nodes can differ at most by one tick. If the timestamps of two events differ by at least two macroticks, the temporal order of the events can be reconstructed.

An **interval** is delimited by two events: start and terminating event.

Given D(obs) the difference between the start event observed by one node and the terminating event observed by another node, if the reasonableness condition is respected, the true duration D(real) of an interval is bounded by 2G:

$$| D(real) - D(obs) | < 2G.$$

Internal Synchronization

In synchronous systems clock synchronization is performed sending the time *t* from a reference server.
Given known message transmission delay upper and lower bounds, the receiving node sets its clock as

$$c = t+(max-min)/2$$

We thus have a max skew of $(max-min)/2$.

Popular protocols for asynchronous systems:

- **Cristian's Protocol**
- **Network Time Protocol (NTP)**

External Synchronization

Possible if the system has access to an external time reference.
GPS is an important time source that gives synchronization accuracy in the sub-microsecond interval.
External and internal clock sync are **complementary**: fault tolerant internal sync provides high availability and short term stability; external sync provides high reliability and long term stability.
A dependable clock shall posses both the characteristics.

Global Navigation Systems

To determine the position Global Navigation System uses triangulation, in space.

$$\vec{s} = \vec{v}\,t$$

The radio signals sent by the satellite travel at the speed of light: ~300,000,000 m/s
We have to measure the travel time of the signal.
Both the Receiver and the Satellite "sing" the same "song": a pseudo random noise. The receiver, is thus able, when it receives the signal to determine the time that the satellite signal has taken to reach the receiver.
To allow the receiver to discriminate between different satellites, every one "sings" a different song.
We can deduce the position if we are listening to at least 3 satellites (triangulation in space).

Global navigation systems **clock signal** is broadcasted continuously.

Possible attacks against GNS: jamming, meaconing and spoofing.
Meaconing is the interception and rebroadcast of navigation signals. These signals are rebroadcast on the received frequency, typically, with power higher than the original signal, to confuse enemy navigation. Consequently, aircraft or ground stations are given inaccurate bearings.

Large infrastructure SoS already use GPS clock signal for CS clocks synchronization. A defect in the value would seriously compromise the security of the SoS. The number of systems relying on GPS clock increases without considering loss-of-signal consequences (no fallback system). Requirement for a Resilient Master Clock emerges.

Reference: Local clock synchronization via NTP through GPS PPS.

Atomic clocks

An **atomic clock** works like a conventional clock but the time-base of the clock, instead of being an oscillating mass as in a pendulum clock, is based on the properties of atoms when transitioning between different energy states.
An atom, when excited by an external energy source, goes to a higher energy state. Then, from this state, it goes to a lower energy state. In this transition, the atom releases energy at a very precise frequency which is characteristic of the type of atom. This is like a signature for the type of material used. All that is needed for making a good clock is a way of detecting this frequency and using it as an input to a counter.

A GPS satellite has onboard at least two caesium and two rubidium atomic clocks. Rubidium is less accurate but is more stable than caesium.

Resilient Master Clock

Low power consumption, low weight and low cost clock able to provide correct time in absence of GNSS signals. Includes an independent oscillator, GPS receiver, Commercial Off-the-Shelf (COTS) sensors and software clock control techniques devised to provide self estimation of clock quality.
Extends the holdover duration by compensating the local clock deviation.

Hardware:

- GPS module: receive time messages from GPS constellation
- Sensors: acquire environment information such as temperature and pressure.
- Communication interface: for example a NIC.
- CPU, memory and physical oscillator: standard components of any hardware board.

Software:

- OS layer: maintains a local software clock.
- NTP: uses GPS acquired information to discipline local clock.
- CDC: Clock Drift Compensation generate a Pulse Per Second signal when the GPS is unavailable. Compensation based on information from the sensors and priori knowledge of frequency deviation caused by environmental changes on the oscillator.
- R&SA Clock: uses the offset obtained by the sync module to estimate the uncertainty of the time provided by the local clock over time.
- Checker. Checks uncertainty output of R&SA clock before PTP broadcast.
- Master PTP: Precision Time Protocol master to synchronize clocks through the network.

Distributed Systems

“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.” Leslie Lamport

A distributed system is a system composed by a set of **nodes**, connected by a **network**, working together and appearing as a **single** coherent system[1].
Each machine depends on the others for its functionality.
These systems can be **dynamic** with nodes joining, leaving and failing.
A distributed system improves scalability, reliability and availability.

Main **classification** parameters:

- Synchrony degree: from synchronous to asynchronous.
- Failures types: crash, omission, value, timing, byzantine.
- Network characteristics: topology, reliability, assumptions.

A node is modeled as a state transition system (like FSM) having a finite number of possible states. Furthermore, a node has a bunch of neighbors, can send/receive messages and do local computations.

Synchrony classification

A **synchronous system** is characterized by having a **known upper bound** to:

- message **delivery delay**;
- information **processing speed**;
- local clocks **drift rate**.

Mechanisms for **fault detection** can be easily implemented using timeouts. Unfortunately, in real world, is not easy to maintain synchronous property over time in a large scale system. Variable or unexpected working loads always cause asynchrony, thus total synchrony is ideal and valid only in a probabilistic way.

An **asynchronous system** is characterized by the absence of at least one of the synchronous system timing bounds. Often characterized by lack of assumption on time (**time-free**).
Asynchrony introduces non-determinism but has a better application portability.

1 The nodes work together to solve a common goal (managed, acknowledged, collaborative SoS)

Consistency models

Under **strict consistency** a write to a variable by any process needs to be seen instantaneously by all the other processes. A distributed system with many nodes will take some time to propagate the information, thus the model is **impossible** to realize with the current technology.

Linearizability is a correctness condition for shared objects that provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.
In order to achieve linearizability, a system should guarantee there's a "point of no return", after which the whole system commits to the new value and won't go back to revert it. Moreover, this all **appears instantaneous,** i.e. there's no "flickering" period where two process reads two different values). This requires absolutely zero clock skew between the machines.

Implementing linearizability in a distributed system might be difficult and expensive, so there exist other useful models with weaker requirements, such as **sequential consistency** (FIFO), which states that result of the execution of an operation set is the same as if all operations were executed in *some* sequential order by all the concurrent processes.

This can be further relaxed with **causal consistency**, which specifies that only causally related writes must have a particular order.

Distributed Coordination

Each machine in a distributed system has its own clock thus there is **no notion of global physical time**.
The *n* oscillators on the *n* computers runs at slightly different rates (drift), causing the clocks to gradually get out of synchronization (skew).
Many real world applications require a **shared global time**. The **timestamp** of an event E, denoted as T(E), is the event shared global time (physical or logical).

Global time solutions:

- **Physical clocks synchronization.** Periodic adjustment of each machine local clock to achieve a known degree of accuracy. Within the bounds of the provided accuracy we can coordinate the activities. Needed for RT CPS.
- **Logical clocks.** monotonic counters that allows to infer a **relative order** (partial or total) between the events across the distributed system. Based on the **causality** relationship.

Causality

An event A is **causally related** to an event B, written $A \rightarrow B$, if the event A may be the direct or indirect cause of the event B.
Given T(E) the timestamp of an event E, **if $A \rightarrow B$ then T(A) < T(B)** (converse not true).
Causality relation implies the **equivalent** logical relation "**happens-before**" (converse not true).

Causal order is a **strict order relationship**:

- *Irreflexivity*: $A \rightarrow A$ doesn't hold for any event.
- *Asymmentry*: if $A \rightarrow B$ then $B \rightarrow A$ does not hold.
- *Transitivity*: $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

is a strict **total** order relationship if also the connex property holds:

- *Connex*: $A \rightarrow B$ or $B \rightarrow A$ hold

If not total the order is **partial**, meaning that not every event is causally related to another.

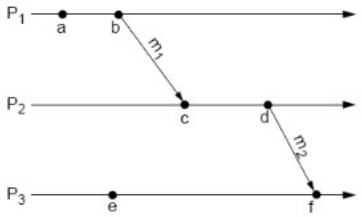
For certain applications, as long as timestamps obey causal order, we don't need physical clocks synchronization.

Lamport's Logical Clocks

Each process P_i holds a monotonic counter T_i used to assign events timestamps.
Rules for process P_i:

- P_i increments T_i before any instruction execution or send event.
- Each sent message carries T_i.
- When receiving a message from a process P_j (containing T_j), T_i is set as $max(T_i, T_j)+1$.

Minor issue: Lamport's clocks impose only **partial order** on the events set. That is, if two events are not causally related then we are not able to establish if one happened before the other.
If two events are not causally related they are classified as **concurrent events** (A||B). Even if they were not really concurrent, neither of the two influenced the other so it doesn't matter which one come first.



$a \rightarrow b \rightarrow c \rightarrow d \rightarrow f$
$e \rightarrow f$
$e||a\ ,\ e||b\ ,\ e||c\ ,\ e||d$

For example, above T(e) < T(c) but they are not causally related. Also note that if A and B belongs to two different nodes, T(A) < T(B) doesn't imply that the local time of A is less than the local time of B.

Total order workaround. For certain applications, events **total order** is required thus we should establish an order **criteria** for parallel events. One possible solution is, given A and B two events happening at processes P_i and P_j respectively, then:

T(A) < T(B) *iff* A → B or (A || B and $i < j$)

Major issue: If A → B then T(A) < T(B) but the converse is not true. In other words, given two events timestamps T(A) and T(B), we are not able to establish if the two events are causally related.

Vector Clocks

Solve both the Lamport's logical clocks issues by adding some extra overhead.
Allow to establish if two events are causally related by looking at their timestamps.
Every event among the processes has a **unique** timestamp value.

Suppose there are *n* processes, each process P_i has a vector V_i of length *n* used to assign timestamps to events. $V_i[j]$ corresponds to P_i's knowledge of latest event at P_j.

Rules for process P_i:

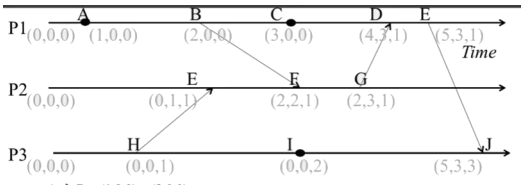
- P_i increments the *i*-th element of V_i before any instruction execution or send event .
- Each sent message carries V_i.
- When receiving a message from a process P_j (containing V_j)
  - $V_i[i] = V_i[i]+1$
  - $V_i[k] = max(V_i[k], V_j[k])$ for $k \neq i$

Results:

- $V_i = V_j$ *iff* $V_i[k] = V_j[k]$, for all $k = 1...n$.
- $V_i \leq V_j$ *iff* $V_i[k] \leq V_j[k]$, *for all* $k = 1...n$.
- $V_i < V_j$ *iff* $V_i \leq V_j$ and there exists *k* such that $V_i[k] < V_j[k]$.
- $V_i \| V_j$ *iff* not $V_i \leq V_j$ and not $V_j \leq V_i$.

In practice there are only two possibilities:

- If $V_i < V_j$ when the two events are causally related.
- If $V_i \| V_j$ when two events are concurrent.



$(1,0,0) < (2,0,0) \leftrightarrow A \rightarrow B$

$(2,0,0) < (2,2,1) \leftrightarrow B \rightarrow F$

$(0,1,1) < (2,2,1) \leftrightarrow E \rightarrow F$

Consensus

p-agreement problem. Given n entities with p entities proposing the same value.

- **Strong majority** : $p = \lfloor n/2 \rfloor + 1$
- **Consensus** or unanimity : $p = n$

Necessary and used for building reliable and secure distributed applications and services. It allows two or more entities to reach a common agreed decision starting from their initial proposed values.
Some important applications: leader election, blockchain, distributed database, virtual machines, etc.

Consensus properties**:**

- **Termination**: every process eventually decides.
- **Validity**: if a process decides for a value v, then v has been proposed by at least one process.
- **Integrity**: every process decides at most one time.
- **Uniform Agreement**: two process do not decide in a different way.

Two phase commit (2PC)

This is a specialized consensus algorithm that assumes a leader and is very used in simple distributed databases contexts. Phases:

- **Voting**: participants dry-run of operations, return the run result.
- **Commit**: only if all votes are positives.

It is a blocking protocol. Participants blocked until a commit/abort is received from the coordinator.
However, if nodes are allowed to fail (even if a single node can fail) then things get more complicated.
The protocol requires the support of a reliable broadcast service.

Best-Effort Broadcast

Eventually delivers the messages ensuring that a message is not delivered more than once. Delivery order is not contemplated. Guarantees message delivery only if sender is correct.
Assumes **network reliability**.
Properties:

- *Termination*: if a correct process broadcasts a message, then all correct participants will eventually deliver it.

- *Validity*: if a process delivers a message then the message was previously broadcast by a process.
- *Integrity*: no message is delivered more than once.

If the sender crashes before being able to send the message to all, some processes will not deliver the message. Thus even termination property is not guaranteed.

Reliable Broadcast

If a sender crashes then all or none correct nodes gets the message.

Assumes **network reliability**.

Properties:

- Best Effort Broadcast properties
- *Agreement*. If a correct node delivers the message, then every other correct node delivers it.

Eager RB. Based on **message diffusion** principle; when a node receives a message it rebroadcasts it and then, if the message has not already been delivered, it delivers it.

Lazy RB. Requires a Failure Detector with strong completeness.

Given a correct process P_i, two cases may happen:

1. Receives a message from P_j, detects P_j failure, re-broadcasts the message.
2. Detects P_j failure, receives a message from P_j, re-broadcasts the message.

The second case may happen because of unpredictable network latency.

Strong completeness is required because if some faulty process are not suspected (weak completeness) the BC is not reliable. On the other hand, weak accuracy is sufficient because if some correct process is suspected then only performances are affected (Eager RB is a Lazy RB where all nodes are suspected).

Uniform Reliable Broadcast

Properties:

- Reliable Broadcast properties
- *Uniform Agreement*: If a node delivers the message, then every other correct node delivers it.

Before delivering a message, a process forwards *m* to all processes.

In absence of communication failures, all correct processes will eventually receive the message and deliver it. The proposed RB algorithm satisfies the URB requirements because of the "*send before deliver*" strategy.

Broadcast Protocols Variants:

- **FIFO**: messages are delivered in FIFO order.
- **Causal**: delivery follows causality relationship (relaxes strict FIFO order).
- **Atomic**: messages delivered in the same *Total Order.*

Note the difference between FIFO and Atomic: FIFO requires that the delivery order is the same of the sent order while the Atomic doesn't impose any particular order. It just requires that the order is the same for all the participants.

Atomic broadcast allows the implementation of **state machine replication** across distributed nodes.

Proposition. *Atomic Broadcast is equivalent to Consensus.*

Proof.

Atomic BC to consensus reduction. A value can be proposed by a process for consensus by atomically broadcasting it, and a process can decide a value by selecting the value of the first message which atomically receives.

Consensus to atomic BC reduction. A group of participants can atomically broadcast messages by achieving consensus regarding the first message to be received, followed by achieving consensus on the next message, and so forth until all the messages are delivered.

FLP (Fisher, Lynch, Paterson) impossibility result. *There is no deterministic algorithm able to solve the consensus problem in a asynchronous system which is able to suffer even only one crash failure*.
(Reference paper: *Impossibility of Distributed Consensus with One Faulty Process*)

The FLP result applies even if we assume the network total reliability and the absence of Byzantine fails. A simple node crash can jeopardize the entire system. The result bottom-line is that in asynchronous systems is impossible to distinguish between a crashed and a slow process.

Atomic broadcast (and thus any consensus algorithm) can be resiliently performed only on Synchronous systems.

Proposition. *Consensus is solvable in synchronous system with up to N-1 crashes.*

Failure Detectors

Failure types :

- **Crash**: a process stops working
- **Omission**: replies arrives infinitely late
- **Value**: incorrect values are sent.
- **Timing**: responses are provided too early or too late.
- **Byzantine**: nodes might behave arbitrarily.

The system has a **distributed oracle**, the **failure detector**, which provides information of possible **crashed** nodes. Each process has access to a failure detector component that monitors the other nodes.

The module has a **dynamic** list of **suspected** crashed nodes. Two FD modules can have two different lists in the same time.
The more synchronous is the system the more accurate the information provided. If is completely synchronous, is possible to have a FD with perfect information.

Typical **implementation**
Periodic exchange of **heartbeat** messages, **timeout** based on worst case message RTT.
If timeout occurs then suspect the node. If receive a message from suspected node, revise suspicion and increase timeout.

Completeness: every faulty node is eventually permanently suspected...

- **Strong**: by *every* correct process.
- **Weak**: by *at least one* correct process.

Is trivially possible to achieve strong completeness by suspecting every node.

Accuracy: every correct node never suspects...

- **Strong**: any correct node.
- **Weak**: at least one correct node.

Is trivially possible to achieve strong accuracy by don’t suspecting any node.

Eventual accuracy: there is a time after that we have strong or weak accuracy.

Because of the **FLP result** it is not possible to achieve strong completeness and strong accuracy in an asynchronous system.

Failure detectors can be partitioned in different classes depending on the Completeness and Accuracy.

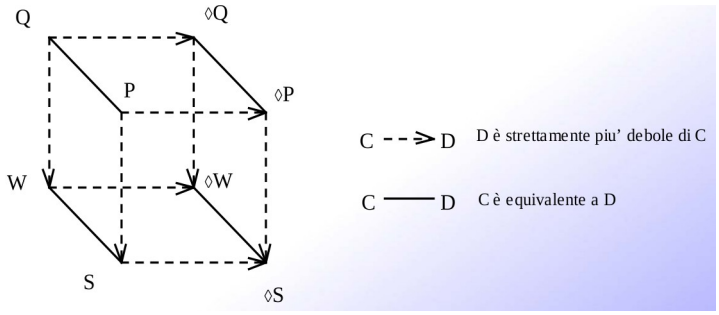
Accuracy / **Completeness**	Strong	Weak	Eventually Strong	Eventually Weak
Strong	Perfect (P)	Strong (S)	Eventually Perfect (♦P)	Eventually Strong (♦S)
Weak	Quasi Perfect (Q)	Weak (W)	Event. Quasi Perfect (♦Q)	Eventually Weak (♦W)

FD with **Perpetual Accuracy**: P, Q, S, W

FD with **Eventual Accuracy**: ♦P, ♦Q, ♦S, ♦W

FD with perpetual accuracy solve consensus in asynchronous systems with no limits on the number of failed processes, while FD with eventual accuracy require a majority of correct processes.

If we find a reduction algorithm that transforms a FD *A* into another FD *B* then any problem that can be solved by *B* can be solved by *A*. Follows that *B* is weaker than *A* ($A \geq B$).



♦W is the weakest FD capable to solve the consensus problem in asynchronous systems.

Consensus using Class S

Characterized by:

- Strong completeness: all failed process are eventually permanently suspected.
- Weak accuracy: at least one correct process is never suspected by any other.

There exist at least a node that is suspected suspected iff is really crashed.

Algorithm (time-free)

Composed of three phases.

1. Each correct process executes *n-1* asyncronous rounds in each of which broadcasts a vector containing the proposed values collected so far and then waits to receive the vector from any

other process which is not in its FD. In case that a process *p* waits for a message from *q* and *q* enters its FD list then it pass to the next node.

2. Each correct processes agree on a vector based on everyone proposal. The *i*-th position contains the value proposed by process *i* or the null value (vectors intersection).
3. Each correct process decides for the first non null value in his copy of the vector.

Because of weak accuracy there is at least one correct process that is never in the FD list of any other process, and thus a process that will send its proposal on each round.

If a process *c* is one correct process that never enters the others FD list and V_p is the process *p* vector then for an arbitrary correct process *p*:

- At the end of Step 1 $V_c \leq V_p$
- At the end of Step 2 $V_c = V_p$

The two propositions can be formally proven.

Consensus using Class ♦S

Possible if the maximum number of processes that may fail is less than half of the totality of processes.

Based on the **leader rotation** paradigm. During round *r* the leader will be process $(r \bmod n)+1$.

If the chosen leader is in the FD then is skipped. The communication happens between the leader and the other processes. Every correct process performs in its turn the role of coordinator to determine a value that can be chosen among the various proposals. If the coordinator is correct and is not suspected by the correct processes then it will succeed in identifying such a value and will then perform a reliable broadcast of such value.

Four phases:

1. Each process sends to the leader its own estimate of the value that should be decided (its proposal) with a timestamp indicating the round in which such an estimate has been taken.
2. The leader collects the majority of such estimates, selects the one with the biggest timestamp and sends it to every process proposing it as the new estimate.
3. Each process *p* has two possibilities:
    1. checking the FD module suspects the crash of the leader and send a nack to it.
    2. sends to it an ack to indicate its adoption of the suggester value;
4. The coordinator waits: $\lceil (n+1)/2 \rceil$ responses (ack o nack).
5. If all the responses are positive the coordinator knows that a majority of processes has changed their estimation adopting the proposed value, consequently it sends (through an R-Broadcast) the request to decide according to this result, which is then done by every process when executing the R-delivery of such proposal.

Early Consensus

Alternative to class ♦S consensus. Rotation of the leader but message exchange is simplified.

The leader proposal is not followed by ack/nack to reach a decision. If *p* receives a value *v*, it forwards it to every other process. By doing so, if *p* receives the same value from the majority of the processes the decision is already taken. If the coordinator is not suspected the protocol terminates directly at the first round.

If the coordinator crashes or is suspected by a majority of the processes, all the processes move to the next round after having updated the estimates, ensuring so that if some process has decided for the estimated of the coordinator at round *r* every process that moved to round *r*+1 has this value as its own estimate. This way the Uniform Agreement property is satisfied.

Reference: *Unreliable Failure detectors for Reliable Distributed Systems* (Chandra and Toueg).

Timed Asynchrony

Asynchronous systems do not guarantee an upper bound to **communication** and **scheduling** delays. Processes cannot correctly distinguish between a crashed and slow but correct process. Fault tolerant services for asynchronous systems must be **timed**.

The **specification** of services offered by these systems describes not only the state transitions and the output in response to requests operations but also the time interval in which such transitions must happen.

This model is in contrast with **time-free** models, where there is the absence of a time reference. In time-free models progression is only triggered by external events.

The termination conditions for time-free asynchronous systems are typically time-free, i.e. they require that an algorithm terminates in a finite number of steps. The termination conditions for synchronous systems are in general time bounded in the sense that they require that operation executions terminate in a bounded amount of time.

The termination conditions for a timed asynchronous system are (conditionally) timed; in an always eventually majority-stable system they have the form: when a process *p* is majority-stable in an interval [t, t+E], then an operation at *p* started by time *t* must terminate by *t+E*. The termination requirement demands that a timed protocol has only timed termination conditions.

Model assumptions

- Processes may be subject to **crash**.
- Processes have access to local clocks which stay a linear envelope of real time (**limited drift**).
- Communications between processes is realized through an unreliable **datagram** service.
- There is **no limit** on the **failure rate** of communications of the processes.
- All protocol **services are timed**. It is therefore possible to define **time-outs** whose passing determines a time failure.

Required **datagram** service is characterized as follows

- Allows unicast and broadcasting.
- Identifies messages in an univocal way.
- Does not ensure the existence of an upper bound on message delivery delay.
- Allows to define a time-out δ on message transmission (one-way timeout delay) whose choice has an impact on failure rates and on system stability.
- Transmission time of messages is proportional on their size.
- It is subject to omission and timing failures.

Each process has access to a local hardware clock with a **drift rate** limited by ρ. Current quartz technology offers granularities between 1us and 1ns. While the clock drift rate ranges between 10^{-4}

and 10^{-6}. Is assumed that through a calibration mechanism local clocks stay in a linear envelope of real time. Clock crash implies process failure, the converse is not true.

Stability Predicates

The time interval between the occurrence of an event and the termination of its processing is called process **scheduling delay**. Let σ be the time-out for scheduling delays. If a process reacts to each event within σ time is said to be **timely** (no performance failures).
Two processes are **connected** in an interval if are timely in such interval and every message exchanged suffers a max delay of δ (one-way time-out-delay).
The choice of δ is such that we can neglect process scheduling delays σ.
If the majority of processes are pairwise connected then they form a **stable majority**.
A process is **majority stable** in an interval if it belongs to a stable majority.
A system is **majority stable** in an interval if if has a stable majority.

Progress assumptions are the conditions which restrict the pace of the processes and the transmission delay.

Empirically has been observed that distributed systems activity is characterized by long periods in which there exists a majority of stable processes alternating with short periods of instability.

Always eventually majority stable (progress assumption):

- After each instability period system becomes majority stable for at least Δ clock-time units.
- Each process eventually becomes majority stable for at least Δ clock-time units or crashes.

Where Δ is an a priori given constant.
In other words: "*Infinitely often a majority of the processes will be stable for a limited time interval*".

As long as the system remains stable (i.e. failures are below a given threshold) it is able to proceed in its computation in a limited time. Therefore it is reasonable to assume that operations and communications offered by distributed systems are **timely** for most of their life.
When a system is stable it behaves as it is a synchronous system.

Failures and recoveries affecting processes not belonging to the stable majority do not affect consensus from being reached.

Protocols designed for timed asynchronous systems that assumes majority stable progress assumption can work seamlessly in sychronous systems in which less than half of the processes may be crashed.

Failure model is given as part of the system model while progress assumptions are separated.
The separation allows to test the model with different assumptions.

Difference with Failure Detectors Models

The impossibility to implement a Perfect FD in an asynchronous system has been demonstrated.
Failure detectors hide aspects related to time at higher levels of abstraction. The FD is discussed as we are dealing with time-free models but the common FD implementation is indeed using timeouts to detect failures.

Rotating leadership

Assumptions

1. At any instant there exists at most one leader
2. If a system is majority stable in an interval *I*, then for every process *p* belonging to a stable majority of *I* there exists an interval [*s*, *s+LD*] contained in *I* where *p* is leader (*LD* : leadership period).
3. A process knows that is the leader and is not required that the others know who the leader is.
4. The clocks of the processes are synchronized and skew limited by some constant.

Assumption 4 allows all processes to define a time grid in which to allocate for each process a time slot during which *p* has the highest priority to become the leader.

Algorithm

1. At the beginning of each time slot each process is a candidate to be elected.
2. Each process is associated with a priority and the election protocol ensures that only the highest priority process is elected.
3. For a process to become a leader it needs to receive a majority of replies to its candidacy and that these replies come in time.
4. After sending its application each process waits for a period of time to receive the candidacies of the other processes before responding to the application with highest priority.
5. After becoming the leader, a process remains as such for LD clock-time units.

This protocol guarantees that when the system is majority stable every majority-stable process will have the highest priority in one of the elections, so everyone will eventually become leader.

The main reason this problem is solved in timed systems is the presence of local hardware clocks that evolve into a linear real-time envelope.

If these were not available, it would not be possible to communicate by-time (i.e. the association of information content over time) and then to determine an upper limit on the delay of messages or to ensure that a process is no longer leading in an instant known to all other processes.

Consensus

The protocol prefers safety above liveness.

When a process starts for the first time its state goes from down to up. When a process starts after a crash its state goes from down to the intermediate restarting state.

The idea is that a leader proposes a value that is not in conflict with any previous decision.

When a process *p* becomes a leader, it first performs a broadcast to know if any other process has already reached a decision or is aware of a previous proposal.

Only a correctly running process UP state will respond to this request for information.

If at least the majority of processes reply to *p's* inquiry, *p* will propose the value of the most recent proposal. When none of the replying majority knows of any previous proposal, *p* proposes its own initial value. If he does not know of any decision or does not receive a sufficient number of answers, he will not take any action.

The leader then sends his proposal indicating his priority with it.

Each process stores the value and priority of the proposal most recently received in a protocol state. Since each leader has a higher priority than all of its predecessors it is easy to establish the most recent proposal.

The leader is allowed to decide on a value *v* only if it knows that a majority of processes know that it has proposed *v*.

The leader therefore first broadcasts its proposal containing *v*, when a majority of the processes acknowledge the proposal, he decides on *v* and then broadcasts *v* in a decision message.

A very important invariant of the protocol is that a majority of processes know the proposed value *v* when the leader decides for it.

A process *p* that performs restart must re-initialize its protocol state before moving to the UP state. A state reinitialization implies the lost of the previous decision (if were taken).

A process can only change its state from decided to not-decided if it crash and restarts.

Because is assumed that the majority of processes are up, thanks to the algorithm, the majority of the processes remembers the value he has proposed before the crash. This will allow him to recover the value when it will become the leader or when he will receive someone else decision or proposal.

Reference:

On the Possibility of Consensus in Asynchronous Systems (Christof Fetzer and Flavu Cristian)

Blockchain

After the agreement information is stored in a chain of **blocks** linked one to each other.
Each block is composed by some **transactions**.
Properties: immutability, non-repudiation, data integrity.
With Blockchain the study of consensus, BFT and its variants had a huge new impulse.
The choice of the consensus protocol impacts on the security and scalability of blockchain.
The network topology is decentralized and could be permissioned or permissionless.

A **block** is the base structure of Blockchain and it is composed by Header and Body.
Header main fields:

- **hash** of previous block;
- **timestamp** of block creation;
- **nonce**: a pseudo random number used by the consensus algorithm.

The body contains the set of transactions often organized as a hash tree (Merkle tree).

Bitcoin Proof of Work

The process to reach an agreement is often called **mining**.
First introduced by Bitcoin cryptocurrency, it is a **permissionless** consensus protocol.
Decentralization permits in all nodes, independently, to verify each transaction.
Transactions go in a pool till they are aggregated in a block and confirmed from miners. A new block of transaction is confirmed by the first miner able to solve a computational expensive puzzle. The solution consistutes the Proof of Work (PoW). The miner able to provide the PoW receives a fee.

In methods such as this, finality is not guaranteed in the blockchain, and when it comes to FLP impossibility, safety was given up for liveness.

Accidental fork. Two nodes give a PoW for two different blocks almost at the same time. In such a case the blockchain is splitted in two branches. In this situation, to add a new block, some nodes will start mining above the first fork, while others above the second. The probability that two nodes give again another PoW at the same time above the two forks is even lower and decreases exponentially as the fork grows. As soon as one chain becomes the longer, the nodes working on the other will leave it in favor of the longer. Transactions on the abandoned fork are (re)submitted for confirmation.
Follows that a transaction confirmation is only probabilistic.

51% issue. If there is a node with enough work power to create a chain longer than the current one, then its chain is the valid one and the old one can be invalidated at any time.

Mining algorithm

Nodes add a coinbase transaction (generate Bitcoin for themselves). Header block is built. Miner find a nonce that, put at the end of the block, has a SHA256(block||nonce) less than a given value. The smaller is the given value the higher is the puzzle complexity. Value begins with a lot of zeros thus is a computationally expensive problem to solve.

Reference: Blockchain consensus

Ethereum Proof of Work

ETHash is similar to the Bitcoin PoW, but the math puzzle required to be solved is not CPU bound but is memory bound (memory hard). This allow more fairness between the miners because ASIC are not (yet) created to break memory access speed barriers.
Reference: ETHash memory hardness explained

Proof of Stake

Nodes that want to participate to the mining process has to buy some cryptocurrency (**a stake**) to have the **probability** *p* (proportional to the stake) to participate to the block validation process.
The algorithm selects with probability *p* the validator, in a way that nobody is sure to participate before mining process starts. If a selected validator is offline, a new one is chosen.

The idea was that if you owned a PoS network's token, you had an interest in the success of that network. The more of the token you owned, the more you had "at stake" if the network is attacked. If the network was successfully attacked, the value of your tokens was likely to significantly drop.
Under this logic, it made sense to grant validation rights proportional to the amount of stake you had in the network. For instance, if your staked tokens represent 10% of all tokens that are collectively staked by validators, you can expect to propose and validate ~10% of all blocks. With 10% stake, you are allowed to have more influence over the network compared to people with less stake because, theoretically, you have more to lose if you disrupt the network.

A key difference between PoS and PoW is that with PoS systems there is no new coin creation (mining). Instead, all of the coins are created in the beginning, this means the validators must be fully rewarded through **transaction fees**.

Proof of Stake, Chain Based vs BFT

Chain-based PoS

Chain Based prefers liveness above safety. Forks are more likely but transactions confirmation is faster. It is a permissionless network.

It is easier to build a complete new blockchain starting from zero. Sybil attack is easier.

Nothing-at-stake issue. Unlike in proof of work (PoW), it costs a validator nothing to validate transactions on multiple forks. It is computationally inexpensive to build on every fork because you no longer need PoW to create a block. Second, validators are expected to build on every fork because it is theorized that it is in their financial self-interest to do so. If validators stake on both (or more) chains, they will collect transaction fees on whichever fork ends up winning. This strategy will be disruptive to consensus and could leave the network more vulnerable to double spend attacks.
Ethereum's Casper aims to take the potential of the nothing at stake theory seriously. In order to reduce the likelihood that validators builds on all forks, validators will be penalized by losing a portion of their security deposit.

Another issue is the **long range attack**.

Sybil attack. The attacker subverts the network by forging a lot of fake identities.

BFT PoS

BFT prefers safety above liveness. Tries to achieve quorum but is slower.

The BFT style network is **permissioned**. Validators are randomly assigned to propose a block. The agreement of the block is done through a multi-round consensus process (like PBFT).

Liveness may be compromised preventing consensus. Censorship attacks may be possible.
Among other considerations, this method of establishing consensus requires less effort than other methods. However, it comes at the cost of anonymity on the system.

Proof of X

There may be a proof for anything valuable. For example:

- Proof of deposit: miners lock an amount that cannot be spent during mining.
- Proof of coin age: stake is weighted by the age of possession.
- Proof of identity: miner must cryptographically prove the identity linked to the transaction.
- Proof of capacity: miner power proportional to the space allocated on disk.
- Proof of elapsed time: node waits a random number before is allowed to generate a block
    - Validator with shortest wait time win the lottery and becomes the validator. Depends on the TEE security (not 100%)

Practical BFT (PBFT)

PBFT It's the first algorithm based on BFT used in a blockchain. It's based on deterministic replicas of a server and it is able to tolerate at most **less than 1/3** byzantine nodes. That is, if *f* is the number of byzantine nodes, the algorithm should have at least *3f+1* nodes.

Due to the presence of a leader node, this model follows more of a "**Commander and Lieutenant**" format than a pure Byzantine Generals Problem, where all generals are equal.

The leader chooses the execution order from client's requests, assigning to each request a different round. Each round, called a view, comes down to 4 phases: request, pre-prepare, prepare, commit.

Algorithm Phases:

1. **Request**
    1. A client sends a request to the leader node to invoke a service operation.
    2. The leader node assigns a number to the requests.
2. **Pre-Prepare**
    1. The leader sends a pre-prepare message inserting the view *v* and message *m*.
    2. The leader inserts the message *m* in its log.
3. **Prepare**
    1. The replica accepts a request if the algorithm is in view *v* and if can verify the message.
    2. Each replica sends to all a prepare message.
    3. Each replica collects messages till it has one pre-prepare message and at least *2f* prepare messages that agree for view *v* and message *m*.
4. **Commit**

1. Each replica sends a commit message in which affirms that has a **quorum certificate** and add all in a log.
2. Each replica collects till it has *2f+1* commit messages for view *v* and message *m* from different replicas.
3. Each replica executes the request, eventually after executing all requests with lower sequence number.
4. Replicas send reply to the client.

Complexity: exponential in number of messages exchanged.

One of the primary advantages of the PBFT model is its ability to provide **transaction finality** without the need for confirmations like in PoW. If a proposed block is agreed upon by the nodes in a PBFT system, then that block is final.

PBFT Variants: optimistic, randomized, XFT, hybrid.

Real Time Systems

A real-time system is a system which behavior correctness depends not only on the logical results of the computations, but also on the **physical time**, when these results are produced.

Classification

Classification on **external requirements**

- Hard real-time vs soft real-time
- Fail-safe vs Fail-operational.

Classification based on **implementation**

- Guaranteed timeliness vs best-effort
- Event triggered vs time triggered

Deadline: instant in the timeline when a result has to be produced.

- **Soft** : misses are tolerable but there is a QoS degradation. The result has utility after deadline.
- **Firm**: infrequent misses are tolerable. The result has no utility after the deadline.
- **Hard**: A miss is not tolerable. Catastrophic events could result after the deadline.

Fail-safe system: there is a state that can be reached in case of system-failure. Requires high **error detection coverage**. (e.g. shutdown triggered by UPS).
Fail-operational system: does not allow to identify a safe state thus, in case of failure, it must continue to be operational (e.g. flight control). Should provide a minimum level of service even after the fault activation.

Guaranteed timeliness : temporal correctness is supported by **analytical** arguments.
Best-effort : guaranteed timeliness relies on **probabilistic** arguments.

For guaranteed timeliness there must be **sufficient computational resources** to handle the peak load and fault scenario. Resource adequacy is expensive, but in hard RT there is no alternative.

State is a condition that persists for an interval of real time, and **event** is an occurrence at an instant.

- **State information**: informs about state attributes at the moment of observation (sampling)

- **Event information**: informs about the difference in the attributes immediately before and after the event. Only consequences of an event can be observed.

A **rare event** occurs very infrequently and in most cases are not covered during workload testing.
In some applications the quality of a system depends on the predictable performance in rare event scenarios.

Event triggered: control signals derived solely from occurrence of events (e.g. interrupt)
Time triggered: control signals derived solely from progression of time (e.g. polling)

Temporal requirements

Temporal accuracy of RT data: time between observation and output (e.g. “display”).
Response time: interval between a *stimulus* and the *response* (in RT systems must be bounded).
Predictability: temporal behavior predictable even in rare event scenario.

Variability of the observation-output delay is a form of jitter at the application level.
Jitter in control loops causes a degradation of measurements quantity. For a jitter Δd the measurement error ΔV is $\Delta V = dV/dt \cdot \Delta d$ (integral of the measured value function in Δd).

Execution time of an event-triggered protocol between two tasks depends on several aspects: task scheduling on sender, buffer management on sender, data-link protocol, media access strategy, buffer management on receiver, task scheduling on receiver.

Architecture

A good interface must be precisely specified to **hide irrelevant details**, lead to **minimal coupling** between the interfacing subsystems (e.g. use the subsystems with generic interfaces), conform to established **architectural style**.

Solid RT systems require a **documented technical system architecture**. Construction is supported by a **framework**. Allows a system to be **elegant:** understandable, maintainable, extensible, cost-effective.
A **documented** technical system architecture allows to **share the knowledge** of a system and prevents the ***don’t touch it again*** issue. Trial and error system building, without a solid and testable architecture, is a dead-end road.

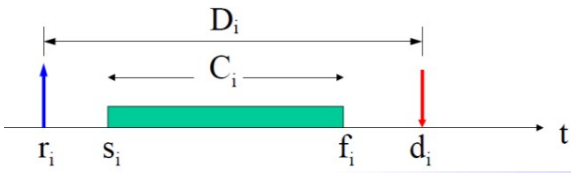
The challenge to manage the complexity of large scale real-time systems is the management of ever-growing complexity. We should **partition** the system in **subsystems** that can be understood and tested independently.

Composability: the architecture framework provides rules for systematic construction of a system out of subsystems (components).
An architecture is **composable** with respect to a given **property** of a sub-system if this property also holds at the system level after the integration. Build systems constructively out of prevalidated components. Example properties where we want composability: **timeliness** and **testability**.
Two level design: separate architecture design from component design.

Scheduling

A **task** is a sequence of instructions that in absence of other activities is continuously executed by the processor until completion.
In a generic multitask OS, a task has a **state**: e.g. ready, running, blocked.
Tasks are kept in the **ready queue** and the next task to be run is decided by the **scheduling algorithm**.
The algorithm is **preemptive** if the running task can be suspended; e.g. to execute a more important task. When a task is stopped and another one is started a **context switch** is performed. The time dedicated to a process before a context switch is called a **time slice**. A **schedule** is a particular assignment of tasks to the processor.

r_i : request time (arrival time)
s_i : start time
C_i : worst case execution time (WCET)
d_i : absolute deadline
D_i : relative deadline
f_i : worst case finishing time



A task set is **feasible** if there is a schedule such that no task misses its deadline.
RT tasks are classified depending on their **deadline**: Hard, Firm and Soft.

Task activation:

- ***Time driven***: periodically activated with period T_i (often $D_i = T_i$).
- ***Event driven***: arrival of an event or via an explicit activation procedure.

Constraints

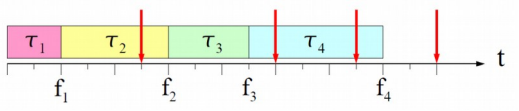
- **Timing**: deadline, activation, completion, jitter. Can be *explicit* (in the system specs) or *implicit* (must be respected to meet the requirements).
- **Precedence**: a prestablished execution ordering exists.
- **Resource**: enforce synchronization for mutually exclusive resources (e.g. wait queues).

Earliest Due Date

(*aperiodic*)
Assumptions: all tasks arrives simultaneously, fixed priority (D_i), preemption not allowed.
Task with the **earliest relative deadline** is selected.
Minimizes the **maximum lateness**: $L_{max}=max\{f_i-d_i\}$). If $L_{max}<0$ then no task miss its deadline

Offline feasibility

The task set is feasible if $\forall i \sum_{k=1}^{i} C_k \leq D_i$



Complexity: $O(n \cdot logn)$ to order the task set and $O(n)$ to guarantee the whole task set feasibility.

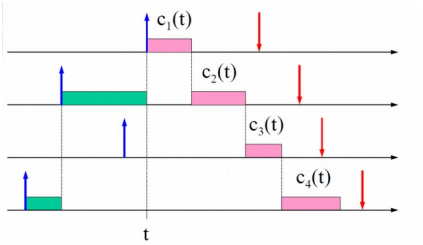
Earliest Deadline First

(*aperiodic*)
Assumptions: tasks may arrives at any time, priority depends on arrival, preemption allowed.
Task with the **earliest absolute deadline** is selected.
Minimizes the **maximum lateness**.

Online feasibility

The task set is feasible if $\forall i \sum_{k=1}^{i} c_k(t) \leq d_i - t$, with *t* the instant where all the tasks are running.

From time *t* the diagram is similar to the EDD case.



Complexity: $O(n)$ to insert a new task in the queue and $O(n)$ to guarantee the new task feasibility.

Cycle Executive

(*periodic*)

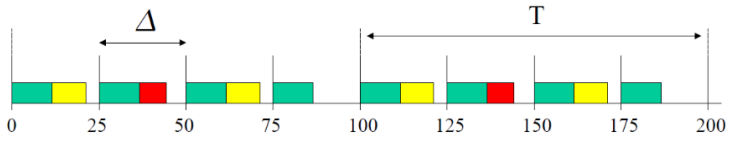
Given a period T_i, with that T_i we must guarantee that each periodic job τ_{ik} is activated at $r_{ik}=(k-1)T_i$ and completes with $d_{ik}=r_{ik}+D_i$.

Offline feasibility

Time axis divided in slots of equal length. Each task is **statically allocated** in a slot in order to meet the desired request rate. Execution in each slot activated by a timer.

Each task has its own period of execution: T_i

task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms



$\Delta: gcd(T_i)$ is the **minor cycle** ,

$T: lcm(T_i)$ is the **major cycle**

Major and minor cycles must respect the WCET guarantees.

Advantages: simple implementation, no RT OS is required, low run time overhead, allows jitter control

Disadvantages: not robust during **overloads**, difficult to **expand** the schedule, not easy to handle aperiodic activities, process periods must be multiples of the minor cycle, difficult to incorporate processes with long periods, difficult to construct and maintain, a process with a variable computational time must be split into a fixed number of fixed size procedures.

Overload problems: what to do when a task **overruns**? Continue can have a domino effect, stop can leave system in inconsistent state. In some systems a rollback is better.

Rate Monotonic

(*periodic*)

Each task is assigned a fixed priority proportional to its rate.

Advantages: transient overruns better tolerated.

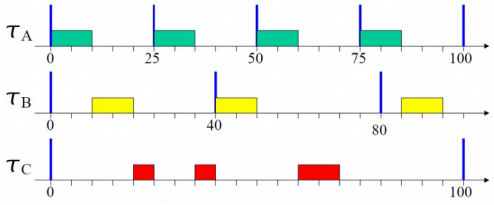
Offline feasibility

Each task *i* uses the processor a fraction of its dedicated (max) time period T_i, thus during its period the processor (max) usage is $U_i=C_i/T_i$.

Given a task set of cardinality *n*, the measure of the processor load is $U_p=\sum_{i=1}^{n} C_i/T_i$.

Necessary condition: If $U_p > 1$ the processor is overloaded and hence the task set is not schedulable. The converse is not true, there are cases where $U_p < 1$ but the tasks are not schedulable by RM.

Sufficient condition: For a *big* number of tasks *n* a sufficient condition is $U_{LOW} = n(2^{1/N} - 1) < 0.69$.



$$U_p = \frac{2}{5} + \frac{2}{8} + \frac{4}{20} \approx 0.85 < 1$$

Execution Time Analysis

Proper **timing analysis** is conducted after implementation: schedulability and WCET.

At design phase only schedules planning with offline feasibility are considered. In general is infeasible to model all possible execution scenarios. Timing analysis is conducted to infer best and worst execution time. WCET is of primary importance in schedulability analysis.

Bounds can be established using methods and tools that considers all possible exec paths of a more abstract, simplified, version of the system. Abstraction looses information, thus estimated WCET should always overestimate the real one.

Empirical measurements

Measuring all execution cases may be an intractable problem.

Execution time computed empirically is **hardware dependent**.

Selected test data may fail to trigger the longest execution path (e.g. exceptions are not triggered).

Combined WCET of smaller parts may not necessary yield the global WCET (composability).

Internal processor state may not be in the worst case **initial settings** (e.g. cached data).

Empirical results may be useful for a first approximation but a more systematic approach may be required to find trustworthy WCET bound on hard RT systems.

Static analysis tools

Static analysis tools do not depend on hardware, there is an abstract **system model**.

Analysis types: values, control flow, process behavior, symbolic simulation.

Control flow analysis: gather information from possible execution paths.

- inputs: task representation (e.g. call graph) and possible additional data (e.g. input data ranges and loops iterations bounds).
- output: constraints of the dynamic behavior of the task