CT420

Real-Time Systems

Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

2025–01–23

Contents

1 Introduction

1.1 Lecturer Contact Information

- Name: Dr. Michael Schukat.
- E-mail: michael.schukat@universityofgalway.ie.
- Office: CSB-3002.
- Name: Dr. Jawad Manzoor.
- E-mail: jawad.manzoor@universityofgalway.ie.
- Office: CSB-3012.

1.2 Assessment

- 2 hours of face-to-face & virtual labs per week from Week 03.
- 30% Continuous Assessment:
  - 2 assignments, 10% each.
  - 2 in-class quizzes between Week 07 & Week 12, worth 5%.

1.3 Introduction to Real-Time Systems

A system is said to be **real-time** if the total correctness of an operation depends not only upon its logical correctness but also upon the time in which it is performed. Contrast functional requirements (logical correctness) versus non-functional requirements (time constraints). There are two main categorisation factors:

- **Criticality:**
  - **Hard RTS:** deadlines (responsiveness) is critical. Failure to meet these have severe to catastrophic consequences (e.g., injury, damage, death).
  - **Soft RTS:** deadlines are less critical, in many cases significant tolerance can be permitted.
- **Speed**
  - **Fast RTS:** responses in microseconds to hundreds of microseconds.
  - **Slow RTS:** responses in the range of seconds to days.

A **safety-critical system (SCS)** or life-critical system is a system whose failure or malfunction may result in death or serious injury to people, loss of equipment / property or severe damage, & environmental harm.

2 The Essence of Time: From Measurement to Navigation & Beyond

Time is the continued sequence of existence & events that occurs in an apparently irreversible succession from past, through the present, into the future. Methods of temporal measurement, or chronometry, take two distinct forms:

- The **calendar**, a mathematical tool for organising intervals of term;
- The **clock**, a physical mechanism that counts the passage of time.

Global (maritime) exploration requires exact maritime navigation, i.e., longitude & latitude calculation. **Latitude** (north-south) orientation is straightforward; **longitude** (east-west orientation) requires a robust (maritime) clock.

Ground-based navigation systems like LORAN (LOng RAnge Navigation) were developed in the 1940s and were in use until recently, and required fixed terrestrial longwave radio transmitters, and receivers on-board of ships & planes. They are also referred to as hyperbolic navigation or multilateration. The principles of ground-based navigation systems is as follows:

1. A **master** with a known location broadcasts a radio pulse.
2. Multiple **slave** stations with a known distance from the master send their own pulse, upon receiving the master pulse.
3. A **receiver** receives master & slave pulses and measures the delay between them.
4. This allows the receiver to deduce the distance to each of the stations, providing a fix.

NEED TO FINISH

3 Time Synchronisation in Distributed Systems

A **distributed system (DS)** is a type of networked system wherein multiple computers (nodes) work together to perform a task. Such systems may or may not be connected to the Internet. Time & synchronisation are important issues here: think of error logs in distributed systems – how can error events recorded in different computers be correlated with each other if there is no common time base? The problem is that GNSS-based time synchronisation may or may not be available, as GPS signals are absorbed or weakened by building structures. There is no other time reference such systems can rely on because in such a distributed system there are just a series of imperfect computer clocks.

In distributed systems, all the different nodes are supposed to have the same notion of time, but quartz oscillators oscillate at slightly different frequencies. Hence, clocks tick at different rates (called *clock skew*), resulting in an increasing gap in perceived time. The difference between two clocks at a given pot is called *clock offset*. The **clock synchronisation problem** aims to minimise the clock skew and subsequently the offset between two or more clocks. A clock can show a positive or negative offset with regard to a reference clock (e.g., UTC), and will need to be resynchronised periodically. One cannot just set the clock to the "correct" time: jumps, particularly backwards, can confuse software and operating systems. Instead, we aim for gradual compensation by correcting the skew: if a clock runs too fast, make it run slower until correct and if a clock runs too slow, make it run faster until correct.

Synchronisation can take place in different forms:

- Based on **physical** clocks: absolute to each other by synchronising to an accurate time source (e.g., UTC), absolute to each other by synchronising to locally agreed time (i.e., no link to a global time reference), where the term *absolute* means that the differences in timestamps are proper time intervals.
- Based on **logical** clocks (i.e., clocks are more like counters): timestamps may be ordered but with no notion of measurable time intervals.

In either case, the DS endpoints synchronise using a shared network. For physical clock synchronisation, network latencies must be considered as packets traverse from a sending node to a receiving node. In a **perfect network**, messages *always* arrive, with a propagation delay of *exactly* d; the sender sends time T in a message, the receiver sets its clock to $T + d$, and synchronisation is exact.

In a **deterministic network**, messages arrive with a propagation delay $0 < d \leq D$; the sender sends time T in a message, the receiver sets its clock to $T + \frac{D}{2}$, and therefore the synchronisation error is at most $\frac{D}{2}$. **Deterministic communication** is the ability of a network to guarantee that a message will be transmitted in a specified, predictable period of time.

3.1 Synchronisation in the Real World

Most off-the-shelf networks are *asynchronous*, that is, data is transmitted intermittently on a best-effort basis. They are designed for flexibility, not determinism, and as a result, propagation delays are arbitrary and sometimes even unsymmetric (i.e., upstream & downstream latencies are different). Therefore, synchronisation algorithms are needed to accommodate these limitations.

3.1.1 Cristian's Algorithm

Cristian's algorithm attempts to compensate for symmetric network delays:

1. The client remembers the local time T_0 just before sending a request.
2. The server receives the request, determines T_S, and sends it as a reply.
3. When the client receives the reply, it notes the local arrival time T_1.
4. The correct time is then approximately $(T_S + \frac{(T_1 - T_0)}{2})$.

The algorithm assumes symmetric network latency. If the server is synced to UTC< all clients will follow UTC. Limitations of Cristian's algorithm include:

- Assumes a symmetric network latency;
- Assumes that timestamps can be taken as the packet hits the wire / arrives at the client;
- Assumes that T_S is right in the middle of the server process; for example, consider the server process being pre-empted just before it sends the response back to the client, which will corrupt the synchronisation of the client.

3.1.2 Berkeley Algorithm

In the **Berkeley algorithm**, there is no accurate time server: instead, a set of client clocks is synchronised to their average time. The assumption is that offsets / skews of all clocks follow some symmetric distribution (e.g., a normal distribution) with some clocks going faster and others slower, and therefore a mean value close to 0.

1. One node is designated to be the **master node** M.
2. The master node periodically queries all other clients for their local time.
3. Each client returns a timestamp or their clock offset to the master.
4. Cristian's algorithm is used to determine and compensate for RTTs, which can be different for each client.
5. Using these, the master computes the average time (thereby ignoring outliers), calculates the difference to all timestamps it has received, and sends an adjustment to each client. Again, each computer gradually adjusts its local clock.

Client clocks are adjusted to run faster or slower, to be synced to an overall agreed system time. The client networks is an intranet, i.e., an isolated system. Therefore, the Berkeley algorithm is an **internal clock synchronisation algorithm**. The Berkeley algorithm was implemented in the TEMPO time synchronisation protocol, which was part of the Berkelely UNIX 4.3BSD system.

3.2 Logical Clocks

Logical clocks are another concept linked to internal clock synchronisation. Logical clocks only care about their internal consistency, but not about absolute (UTC) time; subsequently, they do not need clock synchronisation and take into account the order in which events occur rather than the time at which they occurred. In practice, if clients or processes only care that event a happens before event b, but don't care about the exact time difference, they can make use of a logical clock.

We can define the **happens-before relation** $a \rightarrow b$:

- If events a and b are within the same process, then $a \rightarrow b$ if a occurs with an earlier local timestamp: **process order**.
- If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then $a \rightarrow b$: **causal order**.

- We also have **transitivity:** if $a \to b$ and $b \to c$, then $a \to c$.

Note that this only provides a *partial order*: if two events a and b happen in different processes that do not exchange messages (not even indirectly), then neither $a \to b$ nor $b \to a$ is true. In this situation, we say that a and b are **concurrent** and write $a \sim b$, i.e., nothing can be said about when the events happened or which event happened first.

Happens-before can be implemented using the **Lamport scheme:**

1. Each process P_i has a logical clock L_i, where L_i can be simply an integer variable initialised to 0.
2. L_i is incremented on every local event e; we write $L_i(e)$ or $L(e)$ as the timestamp of e.
3. When P_i sends a message, it increments L_i and copies its content into the packet.
4. When P_i receives a message from P_k, it extracts L_k and sets $L := \max(L_i, L_k)$ and then increments L_i.

This guarantees that if $a \to b$, then $L_i(a) < L_k(b)$, but nothing else.

The primary limitation of Lamport clocks is that they do not capture **causality**. Lamport's logical clocks lead to a situation where all events in a distributed system are ordered, so that if an event a (linked to P_i) "happened before" event b (linked to P_k), i.e., $a \to b$, then $a$will also be positioned in that ordering before b such that $L_i(a) < L_k(b)$ or simply $L(a) < L(b)$; however, nothing can be said about the relationship between two events a & b by merely comparing their time values $L_i(a)$ and $L_k(b.)$: we can't tell if $a \to b$ or $b \to a$ or $a \sim b$ unless they occur in the same process.

3.2.1 Vector Clocks

In practice, causality is captured by means of **vector clocks**:

- There is an ordered list of logical clocks, with one per process.
- Each process P_i maintains a vector $\vec{V_i}$, initially containing all zeroes.
- On a local event e, P_i increments $\vec{V_i}[i]$ (the i^{th} vector component). If the event is "message send", a new $\vec{V_i}$ is copied into the packet.
- If P_i receives a message from P_m, then, for all k