Distributed Systems

Jens Lechtenbörger

Summer Term 2018

Contents

1 Introduction

1.1 Learning Objectives

- Explain “distributed system” and related major notions
  - Definition, examples, goals and challenges
  - Basic scalability techniques
  - Logical time, consistency, consensus
- Contrast synchronous and asynchronous distributed systems
- Compute vector timestamps for events in asynchronous systems and reason about consistency

1.2 Context for Communication and Collaboration Systems

- From a technical perspective, CACSs are distributed systems.
- This presentation is part of four sessions demonstrating more technical aspects of CACSs.
  - Here: **Distributed systems** (DSs) in general
  - Previously: Git as sample DS
  - Upcoming
    - **Internet** as fundamental infrastructure for DSs
    - **Web and e-mail** as sample distributed applications

1.3 Communication and Collaboration

- Communication frequently takes place via the **Internet**
  - Telephony
  - Instant messaging
  - E-Mail
  - Social networks
- Collaboration frequently supported by tools using **Internet** technologies
  - All of the above means for communication
  - ERP, CRM, e-learning systems
  - File sharing: Sciebo, etherpad, etc.
  - Programming (which subsumes file sharing): Git, subversion, etc.
- All of the above are instances of **DSs**

1.4 General Importance of DSs

- **DSs** are everywhere
  - Decentralized, heterogeneous, evolving
  - Variety of applications
  - Variety of physical networks and devices
    - Cloud computing, browser as access device
- IT permeates our life
  - Internet of Things (IoT)
  - From smart devices to smart cities
- How does that really work?
  - Complexity? Functionality?
  - Security? Privacy?

2 Distributed Systems

2.1 Definitions

- A distributed system (DS) is ...
  - Leslie Lamport:
    "one in which the failure of a computer you didn't even know existed can render your own computer unusable"
    - (Lamport is Turing Award winner and (co-) author of seminal papers cited in this presentation)
  - Tanenbaum and van Steen [TS07]: "a collection of independent computers that appears to its users as a single coherent system"
  - Coulouris et al. [Cou+11]: "a system in which hardware and software components located at networked computers communicate and coordinate their actions only by passing messages"

Figure 1: “Internet of Things” by Wilgengebroed on Flickr under CC BY 2.0; from Wikimedia

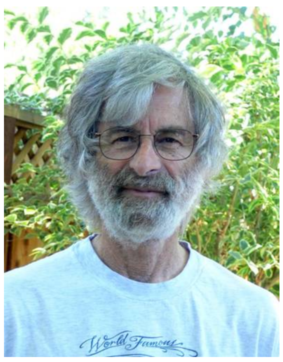

Figure 2: “Photo of Leslie Lamport” under CC0 1.0; from Wikimedia

2.2 Internet vs Web

- (Preview on upcoming sessions)
  - The Internet is a **network** of networks
    - **Connectivity** for heterogeneous devices
    - Various **protocols**
      - IPv4 and IPv6 for **host-to-host** connectivity
    - TCP and UDP for **process-to-process** connectivity (e.g., process of Web browser talks with remote process of Web server)
      - TCP: Reliable full-duplex byte streams
      - UDP: Unreliable message transfer
  - The Web is an **application** using the Internet
    - Clients and servers talking HTTP over TCP/IP
      - E.g., GET requests asking for HTML pages
      - Web servers provide resources to Web clients (browsers, apps)
  - Internet and Web **are** and **contain** DSs

2.3 Technical DS Challenges

- No shared memory but message passing
- Concurrency
- Autonomy and heterogeneity
- Neither global clock nor global state
- Independent failures
- Hostile environment, safety vs security

Recall that in non-distributed systems, within a process its threads share an address space, and processes may share selected regions of memory, which allows them to share data structures as well as to coordinate and cooperate with little overhead. In distributed systems, such sharing and cooperation relies on message passing, adding additional complexity and latency.

Also recall that even in non-distributed systems, concurrency may lead to race conditions, asking for mutual exclusion (MX) and raising various MX related challenges. Clearly, such challenges will also arise in DSs, but they are aggravated by several facts. First, different parts of a system may be run by different autonomous organizations with different goals and different choices concerning hardware, software, and cooperation. Thus, heterogeneity is to be expected and needs to be overcome. Second, we will see that it is already difficult (or even impossible) to agree on such seemingly simple facts as the current time, which led to the development of logical time to avoid the need for globally synchronized time. Similarly, it should not come as a surprise that with multiple autonomous parts, no single party exists that could tell the current global state of a DS. Moreover, different parts of a DS may fail at any point in time (e.g., due to power outage, hardware failure, bugs), but they may also be attacked at any point in time, bringing all issues of single systems related to safety and security to the table.

2.4 DS Goals

- Make **resources** accessible
  - E.g., printers, files, communication and collaboration
- Openness
  - Accepted standards, interoperability
- Various distribution transparencies
- Scalability

  (Source: [TS07])

2.4.1 Distribution Transparencies

- Transparency = Invisibility (hide complexity)
- Sample selection of transparencies from ISO/ODP [FLM95]
  - **Location t.**: clients need not know physical server locations
  - **Migration t.**: clients need not know locations of objects, which can migrate between servers
  - **Replication t.**: clients need not know if/where objects are replicated
  - **Failure t.**: (partial) failures are hidden from clients

2.4.2 Scalability

- Dimensions of scale
  - Numerical: Numbers of users, objects, services
  - Geographical: Distance over which system is scattered
  - Administrative: Number of organizations with control over system components
- Typical scalability techniques
  - Replication, caching, partitioning
- (Scale up vs out)

(Based upon: [Neu94])

As a side note, scaling of hardware comes in two variants:

1. Scale-up (also called vertical scaling), which means to upgrade given hardware (e.g., to add more RAM or more CPU cores)
   - It should be obvious that the potential for scaling up is limited.
2. Scale-out (also called horizontal scaling), which means to add additional machines, often in the form of off-the-shelf PC hardware
   - Here, the potential for scaling out is essentially unlimited.
   - Typically, scale-out is used in combination with partitioning and replication to be explained next.

2.4.3 Replication

- To replicate = to **copy** to multiple machines/nodes
  - Copies (or nodes managing them) are called **replicas**
- **Effects**
  - Increased availability
    - System usable as long as “enough” replicas available
  - Reduced latency
    - Use local or nearby replica
  - Increased throughput
    - Distribute/balance load among replicas
- Challenge: Keep replicas in sync (consistent)
  - Consensus required

2.4.4 Caching

- To cache = to **save** (intermediate) results close to client
  - Temporary form of replication
- **Effects**
  - Reduced load on server
  - Increased availability and throughput as well as reduced latency as with replication
- Challenge: Keep cache contents **up to date**

2.4.5 Partitioning

- To partition = to **spread** data or services among multiple machines/nodes
  - Each node responsible for subset
  - (Sharding = partitioning in shared-nothing architecture)
- **Effects**
  - Reduced availability (each node is additional point of failure)
  - Reduced latency and increased throughput
    - Each node operates on (small) subset
    - Nodes operate in parallel

2.5 Review Question

Prepare an answer to the following question

- How are replication, caching, and partitioning related to scalability of distributed systems?

3 Models

3.1 System Models

- Distributed systems share important properties
  - Common design challenges
- Models capture properties and design challenges
  - Different **types** of models
    - Physical models
      - Computers, devices, and their interconnections
    - Architectural models
      - Entities (e.g., process, object, component), their roles and relationships (e.g., client, server, peer)
    - Fundamental models
      - E.g., interaction, consistency, security
  - Abstract, simplified description of relevant aspects
    - With different layers of abstraction (next slide)

(Source: [Cou+11])

3.2 Layering

- Use **abstractions** to hide complexity
- Abstractions naturally lead to layering
  - General technique in Software Engineering and Information Systems
  - Alternative abstractions at each layer
    - Abstractions specified by standards/protocols/APIs
  - Thus, problem at hand is **decomposed** into manageable components
  - Design becomes (more) **modular**

3.2.1 Hard- and Software Layers

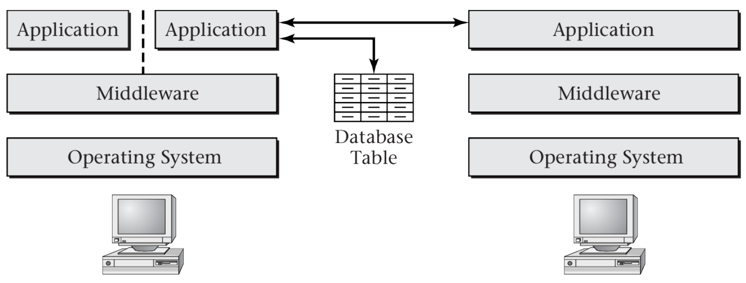

Figure 3: “Figure 1.2 of [Hai17]” by Max Hailperin under CC BY-SA 3.0; converted from GitHub

- OS provides API that hides hardware specifics
- Middleware provides API that hides OS specifics
- (Distributed) Applications use middleware API

3.2.2 Middleware

- Software layer for distributed systems
  - Hide heterogeneity
  - Provide convenient programming model
    - Typical building blocks
      - Remote method/procedure calls
      - Group communication
      - Event notification
      - Placement, replication, partitioning
      - Transactions
      - Security
- Examples
  - ONC/Sun RPC, CORBA, Java RMI
  - Web services via Service Oriented Architecture (SOA) or REST

(Based upon: [Cou+11])

3.2.3 Layered Network Models

- Upcoming session: Layering as core mechanism of network models
  - ISO OSI Reference model with 7 layers
  - Internet model with 4 layers

4 Time and Consistency

4.1 Clocks

- Every computer with own internal clock
  - Used for local timestamps
- Every internal clock with own clock **drift** rate
  - Clocks vary significantly unless **corrections** are applied
- Different correction approaches
  - Obtain time from external source with accuracy guarantee
    - GPS, NTP
- Alternatively, use logical time

Clocks are used to measure passing of time. To that end, clocks produce timestamps, which can be compared against each other to figure out what events happened before or after other events. Clearly, that can only work if clocks used at different places produce (nearly) the same timestamps when read at the same point in time. This slide mentions clock drift as major challenge of physical clocks, correction approaches to address that challenge, and logical time as alternative, which is presented in more detail on later slides.

Before continuing it may be worthwhile to think about what events can be ordered via timestamps. While we may assume time to be totally ordered, under relativistic effects that turns out not to be the case: We may not be able to decide for two events which of them happened before the other (if any). So even in the real world, time only provides a partial order for events.

As we will see, in DSs frequently *logical* clocks are used to assign timestamps to events, and those also only provide a partial order for events. Some pairs of events remain unordered, in which case they are *concurrent*. Often, unordered actions result in arbitrary or even inconsistent outcomes, which points to a need for (a) mechanisms to detect concurrent events (and vector clocks provide one such mechanism) and (b) mechanisms to reach consensus about the desired final outcome, both of which are discussed subsequently.

4.2 Assumptions on Clocks and Timing

- Two extremes
  - Asynchronous
    - Nothing is known about relative timing
  - Synchronous
    - Time is under control, different processes can proceed in lock-step
- April 2018, HUYGENS, [Gen+18]: Time synchronization within tens of nanoseconds based on machine learning
  - (1 Nanosecond = 10^{-9} s)

4.2.1 Asynchronous Distributed System

- **Completely asynchronous** [FLP85]
  - **No assumptions** about
    - relative speeds of processes,
    - time delay in delivering messages,
    - clock drift.
  - Thus,
    - algorithms based on timeouts cannot be used,
    - impossibility to tell whether some process has died or is slow.
- Fits the Internet
  - Uncontrolled resource sharing implies unbounded delays.
  - Solutions for asynchronous systems also work for synchronous ones.

4.2.2 Synchronous Distributed System

- Has **known time bounds** for
  - execution of process steps,
  - transmission of messages,
  - clock drift rates.
- Major strength
  - Algorithms can proceed within **rounds**.
    - For every process, a defined behavior per round exists.
  - **Timeouts** can be used to detect failures.

4.3 Logical Time

- Key insight of Lamport [Lam78]
  - Events can be ordered via "happened before" relation
    - Without reference to physical clock
    - Giving rise to partial order of logical timestamps
- **Happened before**, $\rightarrow$
  1. Each node/process knows order of local events
     - Logical clock produces increasing non-negative integers as timestamps

     (a) Sending of message, event s, must have happened before receipt of that message, event r, denoted by: $s \rightarrow r$

     (b) Transitivity rule: If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

4.3.1 Sample Lamport Timestamps

- Three processes: P1, P2, P3
  - Each process with local clock (initially 0)
    - Clock incremented for each event (including send/receive)
  - Diagonal arrows represent messages
    - Message includes timestamp of sender
    - Receiver computes maximum of sender's and own timestamp, increments result

The graph shown here contains processes P1, P2, and P3 and events occurring in the context of these processes. Time progresses from left to right, and event e_{ij} occurs in the context of process P$\{i\}$. Sample events might indicate completion some algorithmic steps, interaction with I/O, or communication of messages.

The processes are supposed to be part of one DS, but "live" on separate machines. Each machine has its own Lamport clock to produce Lamport timestamps, which are indicated underneath the events.

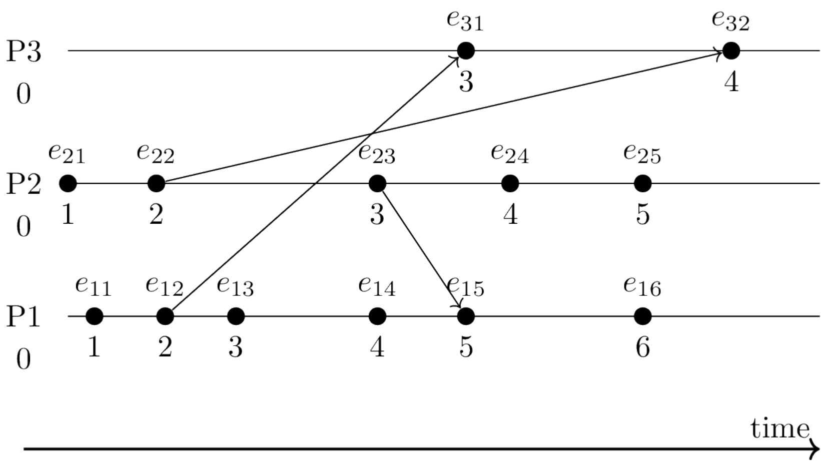

Figure 4: Lamport timestamps for sample events

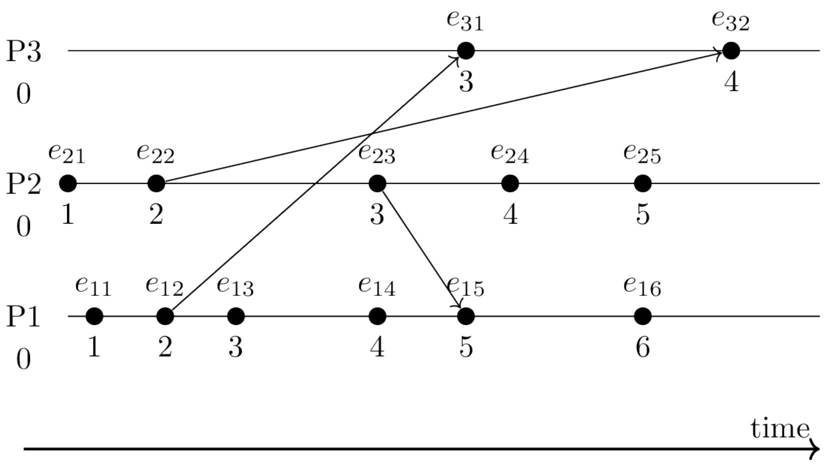

Figure 5: Lamport timestamps for sample events

4.3.2 Lamport Timestamp Properties

- Consider events e and f
  - Let $l(e)$ and $l(f)$ denote their Lamport timestamps
  - If $e \to f$ then $l(e) < l(f)$
    - E.g., $e_{11} \to e_{32}$ and $1 = l(e_{11}) < l(e_{32}) = 4$
  - **However**, if $l(e) < l(f)$ then we cannot conclude anything
    - E.g., e_{32} "last" event but not largest timestamp (4 smaller than several other timestamps)

Intuitively, this slide states that

1. on the positive side the happened-before relation is embedded in Lamport timestamps, but
2. on the negative side, one characteristic of "real" timestamps is missing: Lamport timestamps do not (always) allow us to identify concurrent events.

Vector clocks, presented next, overcome this limitation.

4.3.3 Vector Clocks

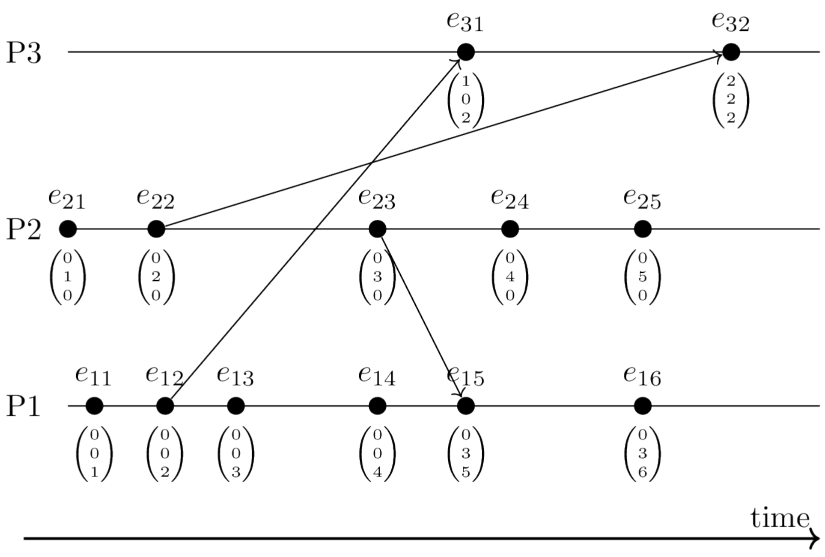

Figure 6: Vector timestamps for sample events

- Vector clock timestamp = vector of logical timestamps
  - Roots in [Par+83], see [RS95] for survey
  - One component per location
    - Incremented locally
  - "Merge" of vectors when message received
    - Component-wise max, followed by local increment

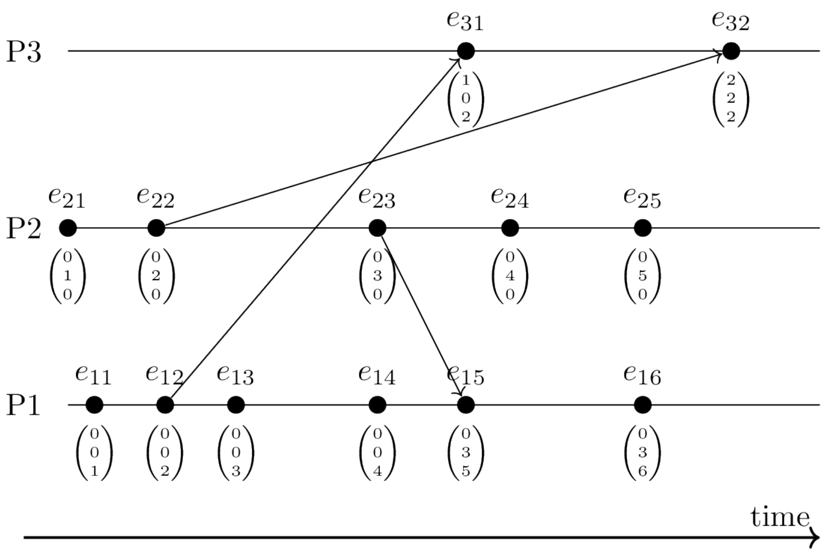

Figure 7: Vector timestamps for sample events

4.3.4 Vector Clocks and Happened Before

- Consider events e and f
  - Let $v(e)$ and $v(f)$ denote their vector timestamps
  - $e \to f$ if and only if $v(e) < v(f)$
    - (Here, "<" is component-wise comparison)
- **Conflicts/concurrency** visible: incomparable vectors
  - Actions at different locations without taking all previous events into account (e.g., e_{23} vs e_{14}; merged at e_{15})

Think about a group exercise assigned to 3 students, namely P1, P2, and P3. In this graph, P1 and P2 start working on a solution in parallel, adding paragraphs to their own, initially empty documents, while P3 is idle.

After adding their first individual paragraphs, both P1 and P2 send their partial solutions (with incomparable timestamps, indicating concurrent/conflicting/unsynchronized work) to P3: When the message from P1 arrives, P3 does not have to do anything special, because the received message has a larger timestamp (which is (0, 0, 2)) than his own (all zeros), indicating that the received message covers all own prior knowledge (which was nothing at all).

When the message from P2 arrives, however, P3 sees an incomparable timestamp (neither of the vectors (1, 0, 2) and (0, 2, 0) is larger than the other). This tells P3 that P1 and P2 worked independently, possibly producing conflicting partial solutions. Now, P3 needs to look at both versions and decide how to merge them. If P3 is lucky, P1 and P2 worked on different sub-tasks, so "merge" would just mean "copy&paste" into a single document; otherwise, P3 might really need to work. Regardless of how this merge is done, afterwards P3 produces the new timestamp (2, 2, 2), which is larger than all other timestamps P3 is aware of, indicating that all prior versions have been integrated.

If you are interested how vector clocks are used at Amazon to manage shopping carts, I recommend that you read this article: [DeC+07]

4.3.5 Review Questions

Prepare answers to the following questions

- Why are Lamport timestamps not sufficient to identify concurrent events?
- How could a continuation of the sample scenario for vector clocks look like such that all shown events are taken into account at all processes? How would the resulting timestamps look like?

4.4 Consistency

- **Lots** of different notions of consistency, e.g.:
  - "C" in ACID transactions: Integrity constraints satisfied
  - "I" in ACID transactions: Serializability
  - All replicas have same value
    * One formal criterion is linearizability
  - **Eventual consistency**: If no updates occur for some time, all replicas converge to the same value
    * Vector timestamps to detect inconsistency
  - Client-centric vs data-centric consistency: See text books
- Consistency requires distributed **consensus/agreement**
  - Next slide

4.5 Consensus

Informal Statement

- Set of (distributed) processes needs to **agree** on value after some processes proposed values.
- E.g.:
  - Who owns a lock?
  - Who is the new master server after a crash of the old one?
  - Who owns a particular Bitcoin?

4.5.1 Byzantine Generals

- Famous consensus example: **Byzantine generals problem** by Lamport, Shostak, Pease (1982) [LSP82]
  - Three or more, possibly treacherous, generals need to agree whether to attack or to retreat
  - Commander issues order, lieutenants must decide
    * Treacherous commander may issue contradicting orders to lieutenants

        - Treacherous lieutenants forward contradicting information to others
    - If not all parties reach the same decision (consensus), the attack fails

- Nowadays, "**Byzantine failure**" is a standard term
    - Arbitrary failure/misbehavior (hardware, software, attacks)

4.5.2 Results on Consensus

- Milestone results; N processes, f of them faulty
    1. [PSL80], synchronous systems: Solutions only if $N \geq 3f + 1$.
    2. [FLP83; FLP85], asynchronous systems: When $f \geq 1$, consensus **cannot** be guaranteed.
    3. [Lam98] (submitted 1990): **Paxos** algorithm for consensus in asynchronous systems
        - State machine replication
            - [Lam98]: "It does not tolerate arbitrary, malicious failures, nor does it guarantee bounded-time response."
    4. [CL99]: PBFT with digital signatures, $N \geq 3f + 1$
    5. [Bur06]: Chubby service (locking, files, naming)
        - Implementing Paxos at heart of Google's infrastructure

5 Conclusions

5.1 Summary

- Distributed systems are everywhere
    - Internet as core infrastructure
    - Networked machines coordinated with messages
    - Various challenges and corresponding techniques

- Asynchronous distributed systems are built without global time
    - Instead, logical timestamps, vector clocks
    - Consensus is standard requirement in lots of scenarios
        - Yet, consensus is hard in presence of failures

5.2 A Different Summary

Warning! External figure **not** included: "Distributed systems" under © 2016 Julia Evans, all rights reserved; from julia's drawings
(See HTML presentation instead.)

5.3 Concluding Questions

- Merge your answers to the following question into our Etherpad or ask them online (Riot or Learnweb)
- What did you find difficult or confusing about the **contents** of the presentation? Please be as specific as possible. For example, you could describe your current understanding (which might allow us to identify misunderstandings), ask questions that allow us to help you, or suggest improvements (ideally by creating an issue or pull request in GitLab).

References

[Bur06] Mike Burrows. "The Chubby Lock Service for Loosely-coupled Distributed Systems". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350.

[CL99] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186.

[Cou+11] George Coulouris et al. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. URL: http://www.cdk5.net/.

[DeC+07] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[FLM95] Kazi Farooqui, Luigi Logrippo, and Jan de Meer. "The ISO Reference Model for Open Distributed Processing: an introduction". In: *Computer Networks and ISDN Systems* 27.8 (1995), pp. 1215–1229. URL: http://www.sciencedirect.com/science/article/pii/016975529500087N.

[FLP83] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Proceedings of the 2Nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. PODS '83. New York, NY, USA: ACM, 1983, pp. 1–7. URL: http://doi.acm.org/10.1145/588058.588060.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* 32.2 (1985), pp. 374–382. URL: http://doi.acm.org/10.1145/3149.214121.

[Gen+18] Yilong Geng et al. “Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 81–94. ISBN: 978-1-931971-43-0. URL: `https://www.usenix.org/conference/nsdi18/presentation/geng`.

[Hai17] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. Revised edition 1.3, 2017. URL: `https://gustavus.edu/mcs/max/os-book/`.

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), pp. 558–565. URL: `http://doi.acm.org/10.1145/359545.359563`.

[Lam98] Leslie Lamport. “The Part-time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. URL: `http://doi.acm.org/10.1145/279227.279229`.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401. URL: `http://doi.acm.org/10.1145/357172.357176`.

[Neu94] B. Clifford Neuman. “Scale in Distributed Systems”. In: *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1994. URL: `http://clifford.neuman.name/publications/`.

[Par+83] D. Stott Parker et al. “Detection of mutual inconsistency in distributed systems”. In: *IEEE transactions on Software Engineering* SE-9.3 (1983), pp. 240–247. DOI: `10.1109/TSE.1983.236733`. URL: `http://doi.ieeecomputersociety.org/10.1109/TSE.1983.236733`.

[PSL80] M. Pease, R. Shostak, and L. Lamport. “Reaching Agreement in the Presence of Faults”. In: *J. ACM* 27.2 (1980), pp. 228–234. URL: `http://doi.acm.org/10.1145/322186.322188`.

[RS95] Michel Raynal and Mukesh Singhal. *Logical Time: A Way to Capture Causality in Distributed Systems*. Research Report RR-2472. INRIA, 1995. URL: `https://hal.inria.fr/inria-00074203`.

[TS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2007.

License Information