# Distributed Systems - Project 2025

## Overview

The goal of this group work is to a) **practise using the methods studies in the course, b) design a complex (and large) distributed system** and then b) **implement a simplified prototype** of it. Typical group size is three (or four) students. The grading is done assuming that the group had three active students. You can do the task also individually or in pairs, but you may have to use more hours per person than in a larger group. The grading focus in on the designed and implemented distribution features and not on the application itself.

The prototype must be running on at least three (3) separate nodes. The nodes can be physical or virtual ones. Each node must have its own IP-address and the communication must be done via message passing; sockets and remote procedure calls (RPC) are both fine. Please notice, that a simple client node used only to test your system cannot be counted in the three nodes.

**The project target is not on the application itself, but using the methods studies on the course.** Your distributed application's functionality can be anything. Some examples include a multi-player game, a collaboration tool, distributed user interface, distributed data processing for sensing devices, etc. Each application instance on a separate node must exchange data with at least two other instances to form a shared state across the distributed nodes.

Some topic examples:

1. A multiplayer game
2. Toy version of content delivery system.
3. Simplified chat service without central service node.
4. Synchronized music player.
5. Your own idea. (end of document contains a list copied from similar course)
6. Any webservice, when the designed large scale system has several (at least 10) nodes, and the 3 required prototype nodes are all on the server side and have complex enough communication between them.

Do not focus your work on the application area, fancy interface or functionalities for the client, but the required distribution features and how they support the selected application area.

## About the distributed application

When selecting the topic, keep in mind, that the nodes need to communicate with each other to provide the service. If implemented fully your application should cover all of the following viewpoints:

a) stateful, the system must have a global state,
b) data consistency and synchronization,
c) consensus (shared decision),
d) fault tolerance, and
e) scalability.

That is why the system documentation / design plan must explain, how your designed system deals with each viewpoint. Not all topic areas need all of these. In such a case explain why the viewpoint is not necessary in your application.

Please notice that the prototype implementation needs to cover the three first three features (a-c). In case your system does not need either consistency or consensus, then the prototype must have fault tolerance features implemented. Scalability is explained in the documentation.

Your system might have different phases and it might remind a centralized service for some purposes and be very decentralized for some others. It might also change the set of functionalities, if some events happen.

When planning your system, think big, because only then would some of the features be actually needed. So plan for a large-scale service (tens or hundreds of nodes) and then think how you can demonstrate the key features with a tiny, functionally-limited prototype.

**Web services**

For web services the project requirements mean that you need to focus on the server side and make it complex enough. So that the group of nodes forming the server side share the state and have actual need for synchronization and consensus. Typically reverse proxy makes the load balancing easier in these services.

You are not allowed to count the node running the client browser interface, if it only communicates with the www server /reverse proxy. You need at least three nodes on the server side.

**Others (like P2P)**

All distributed applications that are not based on www servers and clients have a large option of architectural choices and communication options. Here the key focus is that each node must communicate with at least two other nodes. The system might not even have a human client. Think about the DNS system as an example. On these applications quite often having a dedicated leader (and election) makes it easier to handle consistency and consensus.

## Requirements for the nodes in the system

The planned system typically has plenty of nodes. The implemented prototype must have at least 3 participant nodes, but can have more (or you can run multiple processes simulating extra nodes on some of these). Each of the minimum 3 participant nodes in the prototype must:

1. be running on a separate computer / virtual machine
2. have its own IP-address
3. communicate with at least two other nodes using only Internet protocol based message exchange (the protocol-level message exchange can be hidden e.g. RPC, message queues)
4. be able to express their state and/or readiness for sessions towards other nodes

If your design needs, you can have e.g. client node that communicates only with one server node. Because it only has one communication partner, it cannot be counted to the required minimum number of nodes.

All nodes must log all important events of their activity either locally or to a central location. This feature is useful for debugging and demonstrating the correct functionality of your system.

Nodes can have different roles as long as they meet the requirement of communicating with at least two other nodes. They do not have to be identical. It is very typical in a distributed system that nodes have different functionalities. For example, one can act as a monitor/admin and another one as a sensor/actuator.

Please note that details of session and state are dependent on the application implemented; thus, they depend on design and are not identical across implementations.

## Some technical issues

You can choose programming language yourself. You may even use different languages on different nodes, if you wish.

Grading focuses on the distribution aspect of your system, not on the user interface or application logic. There has to be some application logic and user interface for you to demonstrate the system behavior, but they can be very coarse.

Do not overcomplicate things in the prototype, but do not target too small either – a good initial design will help a lot. Remember to design the communication protocol, that is the messages and how to use them between the nodes.

We recommend you to use virtualization techniques on your own computer at least in the development phase

1. Virtualization technologies can be used for emulating several individual machines on a single computer (e.g., VMWare, Virtual Box)
2. Virtual machines running on a single computer are usually much easier during development than real distinct physical computers
3. When using virtual machines you need to configure the virtual network so that the processes on different virtual machines can communicate with each other.

We also recommend you to use a code repository (preferably GitHub or version.helsinki.fi) during the project to help in the software production and version handling, but also to make the final submission easier.

For testing (and development), you may use the computers: svm-11.cs.helsinki.fi, svm-11-2.cs.helsinki.fi and svm-11-3.cs.helsinki.fi. Please check, by doing the exercise from week 1, that you have access to these.

If you want to use containers as your implementation technique, you can do so, but will not be able to run them on the provided computers. (Containers require that you have root access, which you do not have on the virtual machines we can use.) Please note, that you cannot use one container to implement all nodes, because each node must be placed in a separate physical or virtual computer. With containers the recommendation is to use one container for each separate node and place them manually to different (virtual) machines. Alternatively you can use kubernetes or similar technique for the placement, but that is not required here. With few nodes the manual placement works just fine. If you are not familiar with them already, please do not waste your time in learning containers or kubernetes for this project. It is not needed here. Containers and their placement to nodes with IP must be explained very clearly in the documentation. Documentation in code is not enough.

Simple python, C or java application using datagrams in communication between the nodes works very well for this project.

## About the required prototype functionalities

To be able to gain full points the implemented prototype must have

1) Shared distributed state
2) Synchronization and consistency
3) Consensus
4) And, if one of the above is missing, then fault tolerance

.

**Shared distributed state**

Shared distributed state means that each node maintains some information about the previous events for correct functionality. If node can react to each arriving message without any former information about the history, then the node does not have a state.

**Synchronization and consistency**

Synchronization and consistency have a close relationship with system state, but they focus specifically on the data maintained by the whole system. There are multiple different levels of consistency for data as well as the synchronization between nodes. The required level is application-specific. Some applications tolerate lower levels of synchronization and/or consistency than some other applications.

These have to be addressed in all distributed systems. it is quite typical to try to avoid the need for these as much as possible.

**Consensus**

Consensus is a joint decision-making mechanism, where all the nodes agree on something. It is a distributed process where all nodes participate. If the system has a single, central decision maker (e.g. a master node) then there is no need for consensus protocols about different decisions. Instead, the nodes typically have to agree about the master node using leader election.

**Fault tolerance**

Fault tolerance is an attribute of a system and it covers issues related to failure of one or few nodes, not the whole system. It defines how the remaining nodes handle such a situation and how the system might recover from such an incident. (The recovery of the node back to normal operation is not in the focus of this course, but you could consider e.g. node/process rebooting/restarting and joining to the system.)

**Scalability (and performance)**

Scalability and performance of your system has to be addressed in the final report. These are not part of the implementation, but a discussion topic in the document. Scalability should have been part of the design phase, but not in the focus of prototype implementation. Here the goal is to discuss the system also from these viewpoints, e.g. explain the limitations related to scalability, or clarify the scalability and performance issues of your application in general.

## Deliverables and grading

You need to submit four deliverables (design plan, video or live demonstration, program code and a report)

The reports (design plan and final report) must have a cover page with project name, group number and team member names. Final report must also have table of content.

**Design plan** ( 5 points)

**- THINK BIG! - explain your system as it would be if used globally and large scale.**

The goal of the design plan is to start working with your programming project early during the course. Plan gives you maximum of 5 points. You will get one point for each of these elements

- names of team members and one paragraph description of the selected topic and key points related to the required features
- at least one figure explaining the top level (architectural) view of the nodes, their roles and communication between nodes
- more detailed description of the topic and/ or selected solution techniques/methods
- description of different nodes, their roles and functionalities
- description of what messages the nodes need to send and receive to be able to perform their operations.

Design plan is submitted as pdf-documentation in the moodle.

**Video or live demonstration** ( 5 points, maximum duration of video 5 minutes, presentation duration maximum will be somewhere between 5 and 10 minutes, depends on the number of presentations)

We will organize two demonstration session, where the groups can present their projects. Alternatively to live presentation groups can provide a video. Video or on-site demonstration must demonstrate the system setup and all key functionalities.

Points 1-5 will be given based on the coverage of features in the demonstration/video, as well as clarity of the demonstration. The demonstration clarity does not contain fancy features or technical video editing features, but the flow of presentation, expectations of the audience knowledge level, etc features of a good oral presentation.

**Source code in repository** (5 points)

Source code must be submitted for review in a repository (preferably GitHub or version.helsinki.fi).

The code must be readable. This usually requires comments and good naming conventions with clear and meaningful names.

The code is graded with points 1 to 5 depending on the clarity of the code. Also good coding practices play a role in the grading.

Note that the teachers of the course must be able to access the repository content.

Code grading points can be lower, if the connection between the explanations in the final document and the code are not immediately clear and teachers need to search for them in the code.

**Final report** (15 points)

Final report must be in line with design plan (all major changes must be documented as appendix), program code and the video. If there are issues in this, it will be notified in the grading of the final report, code or video.

In addition to cover page and table of content, the final report must address:

1. The project's goal(s) and core functionality. Identifying the applications / services that can build on your project.
2. The design principles (architecture, process, communication) techniques.
    - This is typically the main part of the report, because it documents the system design and maps it with the source code
3. What functionalities and how does your system provide? For instance, global state, consistency and synchronization, consensus, fault tolerance and recovery, etc? For instance, fault tolerance and consensus when a node goes down.
4. How do you show that your system can scale to support the increased number of nodes?
5. How do you quantify the performance of the system and what did you do (can do) to improve the performance of the system (for instance reduce the latency or improve the throughput)?
6. The key enablers and the lessons learned during the development of the project.
7. Notes about the group members and their participation, work task division, etc. Here you also may report, if you feel that the points collected to group should be split unevenly among group members. Use percentages when descripting this balancing view point.

The grading of the final report will be split among the topics so that parts 1&2 together will give you 1-8 points and the parts 3 to 6 each 1-2 points. (Note this totals to 16, but maximum is 15)

Please notice that some elements (like design principles, scalability etc.) should also be explained in video/demonstration or commented in the source code. If they are comments in the source code, make it very clear in the final document and explain how they can be found it the source code.

Other elements that play a key role in grading of video/demonstration, source code and final report are

1. A running system that implements the basic goals and functionality
2. Functionalities provided by the system.
3. Demonstration of system scaling. For instance, begin with 3 nodes, and then show the system can support 4 nodes, 5 nodes, and more than 5 nodes.
4. Evaluation of the performance in term of relevant metrics such as throughput or latency etc. Identifying what can be done to improve the performance.

**Point balancing**

By default the grading of project is even, so every member gets the same amount of points for each element.

If the workload or contribution is unbalanced, the group can add a separate section to the final report and inform teacher how the points should be balanced. This balancing must be informed using percentages. In a balanced situation each group member gets 100% of the points. Team can move 10, 20 or 30% from one or more students to one or more another students, if they feel that some students' contribution is less that some others' contribution. You must not leave unused percentages, but you cannot invent extra percentages either. The total has to be x*100, where x is the number of students in the group who have not dropped out.

The given project points are then scaled based on these percentages in such a way that student with the highest percentage will get the given score and other gain less points in relationship with their percentage. no student can get higher score than the project documentation earns.

Student, who has dropped out of the group, will get 0 points. However, it is possible to get points for the design plan, if the student drops out after the design plan has been delivered. If the group feels that the dropped out student has had feasible contribution to code or final report, then they can recommend a percentage that this student will get from the project points. This percentage can be 10, 30 or 50.

**Suggested Schedule:**

Weeks 1 and 2; form the team and agree about the project topic

Week 3: submit the design plan, decide the programming language, create repository (deadline for design plan Friday 15.11)

Week4: basic skeleton/outline of the code, identification of the strategy to evaluate and demonstrate the system.

Week 5: A basic running system that can work on 3 nodes (at the latest teacher feedback from design plans)

Week 6: Preliminary evaluation, and implementation of the scaling and functionalities

Week 7: Finalize your system and create the final deliverables, live demo sessions, deadline for submission Monday 15.12 (extra time until Tuesday 16.12 possible, but course exam will be on Wednesday 17.12.)

Topic ideas from https://www.scs.stanford.edu/24sp-cs244b/labs/project.html

(Duration for these projects was 12 weeks. We only have 6!)

Here are some ideas you might be interested in for projects. This list is by no means exhaustive.

- Build better tools for remote collaboration (text, voice, or video chat).
- Build something like Porcupine that addresses some of the paper's shortcomings.
- Distributed protocols such as 2PC and Paxos are (1) short, (2) really hard to get right because of failures and uncertainty. Build a simple system that takes an implementation of these protocols and systematically explores their behavior in the face of crashes and network partitioning. See here for an example of how to do this for file systems.
- Build a checking infrastructure than can plug into the many different RAFT implementations and find protocol errors. The nice trick you can use here is that you do not have to specify correctness: each of the protocols must do the same observable action given the same sequence of crashes, partitions, recoveries. You may want to look at what Kyle Kingsbury has done with Jepsen.
- Build a clean, simple implementation of view stamped replication based on the updated Liskov paper that can be dropped into distributed systems in a way analogous to RAFT.
- Raspberry/pi is a very popular embedded computing platform. Build a distributed system using r/pi nodes and some interesting cheap hardware. More ambitious: build a clean, simple "bare-metal" toolkit on r/pi that allows people to easily build such systems.
- Build a simple, automatic distributed-parallel make implementation. Most makefiles are broken with spurious dependencies (slow) and missing dependencies (incorrect). Fortunately you can infer true dependencies automatically: kick off an existing (broken) build, intercept every "open()" system call to see which other files a given file depends on (e.g., all the files it #includes). Build a lightweight distributed system that does parallel distributed builds using these dependencies.
- Build a large file store, like GFS, and possibly using RAID like Zebra.
- Build a scalable virtual disk like Petal. (Maybe built using the Intel Open Storage Toolkit).
- Build a simplified version of a synchronization service like Google's Chubby.
- Build something like MogileFS but instead of having a centralized database, replicate the DB using Paxos/RAFT.
- Build a scalable web cache using consistent hashing or CARP.
- Build a highly-available, replicated DNS server that uses Paxos or RAFT to ensure consistency of updates.
- Build a parallel debugger (ideally using some modification of GDB) that allows you to debug distributed systems. It should follow execution across message send and receive (analogously to procedure call/return).
- Build a distributed profiler that allows you to observe where time really goes in a distributed system. You should use it to spot bottlenecks in at least one existing distributed system.
- Build a system-call or message-level interposition library that can be slipped underneath an existing networked server and transparently be used to replicate these services so that they can survive failure and network partitioning. (Something similar but more complicated that what you would build: parrot.)
- Build a similar message-level interposition library that can be slipped underneath existing networked services and add security (nonces, secure checksums, encryption, authentication). Relevant: VPNs.
- Build a file synchronization tool like tra.
- Design and implement a Byzantine-fault-tolerant version of Raft.
- Design and implement a Byzantine-fault-tolerant state machine replication system that uses witnesses to keep fewer than $3f + 1$ copies of the state, even with $3f + 1$ servers.
- Build a raftscope-like visualization tool for a different protocol.
- Formally model and verify a consensus protocol, e.g., using TLA+ or IVy.
- Build a mobile-phone based privacy-preserving contact-tracing system for tracking the spread of infectious disease.

- Build a replicated system that leverages CRTDs (conflict-free replicated data types) to achieve eventual state conversion.
- Build a blockchain-based key management server that is more secure than the current PGP key servers. Optionally provide increased privacy via a technique like CONIKS.
- Design an asynchronous RPC library or other infrastructure around C++20 coroutines, so as to hide stack ripping while allowing a high degree of concurrent I/O.
- Modify an open-source database to use a public blockchain as a two-phase commit coordinator, so that you can securely commit an atomic transaction across any two systems using your version of the database. See this paper for inspiration.
- Build a replicated database that handles transactions in batches (i.e. for a blockchain), but where transactions in a batch can be applied in parallel and in any order. To maintain replicability, this database needs a set of *commutative* transaction semantics to permit this database to perform useful work (for any definition of “useful”).