CS425, Distributed Systems: Fall 2026
Machine Programming 2 – Distributed Group Membership

Released: Sep 15, 2026
Due (Hard Deadline): Sun, Sep 27, 2026 (code + report due by 11.59 PM)
Demos on Monday, September 28, 2026

The time for this MP is rather short! So please start now!

Please stick with the groups you formed for MP1.

DO NOT use LLMs for generating your MP2 code. You'll lose out on valuable learning of fundamentals. If we find signs of LLM usage or if you're unable to explain your code during your demo, then **you will get a zero for this MP**.

Covfefe! liked your previous work in MP1, so they've commissioned you to build a distributed group membership service for them.

Membership service. The membership service maintains, at each machine in the system (in a daemon process), a list of the group, i.e., node IDs of machines in the group that are connected and up. There is only one group at any point in time. This membership list is **a full membership list** and needs to be updated whenever:

1. A machine (or its daemon) *joins* the group;
2. A machine (or its daemon) *voluntarily leaves* the group; or
3. A machine (or its daemon) *crashes*. You will assume a **crash/fail-stop model**, i.e., a crashed machine does not recover.

The **node ID** is of the format: <*IP, port, timestamp/version_number*>

The timestamp/version_number is necessary because we assume a fail-stop, not fail-recovery, model. Thus, when a machine rejoins (after a failure or a leave), its timestamp or version number must be incremented to distinguish successive rejoins of the same machine. (Don't confuse these timestamps with suspicion incarnations, which are different).

Completeness. A machine failure must be reflected in at least one membership list within 3 seconds (in wall-clock time), no matter the network latencies. A machine failure, join, or leave must be reflected within 6 seconds at *all* membership lists. This is called *time-bounded completeness*. In this MP, **at most three machines can fail simultaneously**. After any set of simultaneous failures, the next set of failure(s) won't happen for at least 15 seconds, i.e., enough time for your system to converge and agree on a good topology. Your system must ensure completeness for all such failures (with up to three simultaneous failures).

Algorithm. You must use the heartbeating style of failure detection. Implement two variants:

1. **Gossip**: Gossip-style heartbeating (as discussed in class), and

2. **Gossip+S**: Gossip-style heartbeating with a Suspicion mechanism (like SWIM, but without any pinging). Note that the original Gossip does not have Suspicion, so you will have to design it carefully.

You will compare these variants experimentally (see Report section). Think of the parameter settings you need (frequency of heartbeats and gossip, timeouts, etc.) to achieve the time bounds mentioned above.

For **suspicion**, you will need to implement incarnations—see lecture slides/video (and also page 7 of the SWIM paper from the website). Note that when Suspicion is enabled, detection time is measured up until when the entry is finally deleted (after suspicion timeout), not when the failure detection timeout first expires. This is consistent with application (on top of membership) expectations of being told about the failure just once, only after it is confirmed (locally). Once the failure detection layer tells (confirms) an application about a failure, it cannot be rescinded.

For failure detection or leaves, **you cannot use a leader**, since its failure must be detected as well. You must use the Gossip-style heartbeating as mentioned above. Also, **do not use ping-ack** style failure detection like SWIM (though you can borrow the Suspicion style from SWIM's design).

How do nodes join a group? To enable machines to join the group, you can have a **fixed, well-known machine** that all potential (joining) members know about, called the "**introducer**". The new node sends a join request to the introducer. The introducer adds the newcomer to its own local membership list and includes it in the next round of gossip. You can assume that the introducer does not fail before the new node information has spread to other nodes. When the introducer is down, no new members can join the group – but the rest of the group should proceed normally; failures should still be detected, and leaves should be allowed. You need not handle the rejoining of the introducer after it fails.

Scalability requirements. *Your algorithm must scale to large numbers of machines*. For your report experiments, you may assume that you have N > 5 machines in the group at any given time. Typical demo runs will involve 7-10 VMs. Your algorithm must use a small amount of bandwidth (defined as Bytes per second, and NOT as messages per second) to satisfy the above requirements.

Message Drops. You must also write code to randomly drop x% of incoming application-level messages at the receiver, based on the drop-rate setting. If a message is dropped, the receiver must discard it and must not perform the normal message-handling logic for that message.

Note that this is an application-level drop simulation. You should not use network-based tools, such as traffic-control utilities, to drop packets or messages. The goal is to simulate message loss from the membership protocol's perspective, not packet loss at the network layer.

Here is how you can implement this:

- For UDP, dropping means ignoring the received datagram.
- For TCP, dropping means discarding the application-level message after reading it from the stream.

- For gRPC, dropping means discarding the received request before invoking the normal handler logic.

Commands for demos.

Implement command-line commands that allow you to do the following at each process P_i

1. `list_mem`: list the membership list
2. `list_self`: list P_i's ID
3. `join`: join the group (it is ok for this command to be implicitly executed when P_i starts, or you could implement the command explicitly)
4. `leave`: for P_i to voluntarily leave the group (different from a failure)
5. `display_suspects`: List all processes P_i ever suspected along with the time at which P_i suspected the process (since P_i launched)
6. `switch ({suspect, nosuspect})` to enable or disable suspicion. You should **seamlessly switch** your system between the two modes, with the same membership list (i.e., **without needing to stop and restart the system**, re-add or reboot nodes, etc.)
7. `display_protocol`: show whether suspicion is enabled or disabled
8. `set_drop_rate` (default = 0%): artificially simulate a specific message drop rate. Note that the drops must occur on the receiver process (of the message) and not at the sender.

In the **demo**, we will ask you to perform sequences of joins, leaves, crashes, or rejoins of specific processes. When we ask you to "kill" (or fail) a node, please **force-kill** or **crash** the process(es) (using kill -9). You should **not** "shut down the VM" or "leave the group" or any variant of it (since none of those are crashes); if you do shut down the VM, then you will lose significant points. Reuse the MP1 demo script to force-kill MP2 on a particular VM. Also, reuse the demo scripts from MP1 to start MP2 processes on all VMs or a specific VM.

Logging. Create logs on each machine, and use MP1 for debugging. **These logs are important, as we will ask to grep through them at demo time.** Therefore, you must integrate MP2 with your MP1. You can make your logs as verbose as you wish, but at the very least you must log both to the local log file (named machine.i.log) **and** to the local terminal on VM i:

1) each time a change is made to the local membership list **(join, leave, or failure)**;
2) each time a failure is **detected** or **communicated from one machine to another**;
3) each time a **node suspects another node** at the VM terminal of the suspecting node.
4) each time your code **drops a message** at the receiver

Finally, each time you log an event, include the **local time at i** as part of the log entry.

Other considerations. Design first, then implement. Keep your design and implementation as simple as possible. Pay attention to the format of messages that are sent between machines. Ensure that any platform-dependent fields (e.g., integers) are marshaled (converted) into a platform-independent format. Make your implementation bandwidth-efficient. We also recommend (but don't require) writing unit tests for each of the join, leave, and failure functionalities. At the least, ensure that these actually work for a long series of join/leave/fail events.

Language: Choose your favorite language! We recommend one of C++/Java/C/Go/Rust (but you are free to go beyond this list!). We will release "Best MPs" from the class in these languages only (so you can use them in subsequent MPs).

Report: Write a report (typed, not handwritten) of less than 2 pages (12 pt font).

A. Design. Write 1-2 paragraphs justifying your design choices, especially

1. How you integrated Gossip and Suspicion?
2. Why does your MP satisfy the 6-second completeness requirement?

B. **Measurements.** Given the worst-case detection time bound for 10 VMs (average detection times within 5% of each other is ok), compare Gossip vs. Gossip+S across the following metrics:

1. Background bandwidth in the no-failure scenario as a function of the number of VMs in the group.
2. False positive rate (number of false positives per second) when there are no failures as a function of message drop rate. Vary the drop probability within a range of small values: 0%, 1%, 5%, 10%, etc., by introducing artificial message drops on the receiving end. Make sure you run the experiment long enough (e.g., not just until the first false positive, which can give you misleading results).
3. Detection times as a function of the number of (simultaneous) failures.

Draw plots – this is more visual than a table or numbers inside text. For each plot:

- Choose at least 5 values on the x-axis.
- For each data point, take at least as many readings as is necessary to get a non-zero false positive rate (at least 5 readings each), and plot averages and standard deviations.
- Standard deviations should always be plotted as "error bars" alongside average. **Do not plot SD as a separate plot from average – it is hard to reconcile the two**.
- Discuss trends, and whether and why they (do not) match your expectations. Measurement numbers don't lie, but we need to make sense of them!

Stay within the page limit – you will lose points for going over the limit!

Evaluation Break-up: Demo [60%], Report (including design and plots) [30%], Code readability and comments [10%].

Submission:

(1) **Report.** Submit your **report** to Gradescope BEFORE the deadline, and **TAG** your page(s) on Gradescope (there will be a penalty if you don't). Only one group member should submit – you must tag your partner.

(2) **Demo.** There will also be a **demo** of each group's MP2. Sign-up sheets for the demo & demo instructions will be posted on Piazza. Signup for demos on the Friday before the deadline.

(3) **Code.**

A. Submit your working **code** into a subfolder named mp2 in your **GitHub** course repo. In your repo, please include a README explaining how to compile, how to run your unit/demo tests, and how to run your MP2 code.

B. Fill out the MP2 **code** submission **Google form**.

Further submission instructions will be posted on Piazza.

When should I start? We strongly recommend that you start early. Each MP involves a significant amount of design, implementation, debugging, and experimentation work. **Do not** leave all the work until the end.

Academic Integrity: You cannot look at others' solutions, whether from this year or past years. We will run Moss to check for copying within and outside this class – first offense results in a zero grade on the MP, and second offense results in an F in the course. There are past examples of students penalized in both those ways, so just don't cheat. As mentioned earlier, **DO NOT use LLMs** for generating your MP2 code. You can discuss only the MP spec and lecture concepts with other students and on the forum, but not solutions, ideas, or code. If we see you posting code on the forum, that's a zero on the MP.

Contributions by Group Members: We recommend you stick with the same group from one MP to the next (this helps keep the VM mapping sane on IT's end), except for exceptional circumstances. We expect all group members to contribute equally to the overall effort. If you believe your group members are not, please have "the talk" with them first, and give them a second chance. If that doesn't work, please email both instructors.

Happy Membership (from us and the fictitious Covfefe! Inc.)!