NYU Distributed Systems: Safety and Liveness

When designing distributed protocols or reasoning about protocols that others have designed we need to think about correctness. When thinking about the correctness of basic algorithms, e.g., sorting, we often reason about their output, e.g., a correct sorting algorithm takes an arbitrary list (or vector) and produces a sorted list.

However, in the context of distributed systems, where multiple processes are executing concurrently and processes can fail, reasoning about the output at a single point in time often does not suffice. For example, consider a distributed protocol `agreement` where each process is provided an input (i_p for process p) and eventually each process must either write an output o_p to its memory or fail. Further, assume that a correct `agreement` protocol guarantees that if two processes p and q that write outputs o_p and o_q, then $o_p = o_q$.[1] Observe that in the description provided above a process p might fail *after* writing output o_p. Thus checking the values output once all processes have finished (or failed) does not suffice, since we would miss out on cases where $o_p \neq o_q$ because p failed after writing its output.

Instead, we need to consider how a distributed system's *state*, that is, the memory of each process that is executing the protocol and the set of messages waiting to be delivered, evolve over time. In the case of `agreement`, this allows us to reason about outputs from all processes, including those that fail after output. Correctness properties specified in terms of how a program (including a distributed system) evolves over time are known as **trace properties**. While we will not discuss the proof here, Alpern and Schneider '85 (which you read for class) shows that any trace property is an intersection of safety and liveness properties, which we discuss below. As a result, throughout this course you can expect to see safety and liveness guarantees for the protocols we discuss, because specifying both essentially allows us to capture all of the *correctness* properties we care about. Of course, this might not suffice for capturing performance, resource efficiency, or other similar requirements.

What is a Trace?

For our purposes in this class (other definitions and descriptions might apply in other contexts), a single trace represents the *sequence of states* a distributed system (a term we use to refer to a particular deployment of a distributed protocol) during an execution. When reasoning about correctness, we will often need to consider the set of all possible traces that can be produced by a distributed system (or more generally from the protocol): The traces in this set are dictated by the protocol (which determines what messages are sent, how they are processed, and state updates), the failure model (which determines how many processes might fail, and how), network assumptions (which determine whether messages can be dropped and the order in which they are received) and assumptions about timing and fairness.

[1] The class of protocol described here is called consensus, and is going to be a very big part of the class in the weeks going forward. The correctness property we have described is called *agreement*.

State

As we said above, a trace consists of a sequence of **states**. In this case the state of a distributed system is given by

- The state of each process: this can either be what is in their memory, that is the value stored in each variable, or a designate value indicating that the process has failed.
- The set of pending messages, that is the set of messages that have been sent but not received (nor dropped).

If a trace captures an entire execution, then the first state in a trace (that is the first state in a trace) will be one where:

- No process has failed.
- Each processes state is the initial state (e.g., all integers are set to 0).
- No messages are pending.

Transitioning from one state to another

Given a trace (e.g., a trace containing only the initial state) we can extend it by adding another state to the end of the trace. To do so we first need to *select an event* from the set of **enabled events** in the trace's last state. An enabled event is one that can be **applied** to a state. The set of enabled events for a trace ends in state S are:

- The delivery of any message m that is in the set of pending messages, and is intended for a process that is alive and whose delivery does not violate the network assumptions (e.g., does not affect ordering guarantees).
- A timer going off at a process. Note, by convention we consider timers to always be enabled for all processes. However, a protocol might not specify how a process handles a timer, and in this case we treat a timer a no-op, that is, it does not change the state of the process nor does it send any messages.

Once an enabled event has been selected, it can be applied to the last state in the trace and the result appended to the trace to extend it. Repeating this process several times allows us to extend a trace by an arbitrary amount.

We say that a state S is reachable from a trace T if it is possible to extend T to a trace T' (by adding 0 or more states) so that the last state in T' is S.

Finally, because this will be handy below, given a trace T for an execution, and a trace σ we say $T\sigma$ is valid if the first state of σ is reachable from the last state in T.

Trace Properties

Finally, we will talk very briefly about the two trace properties in the reading: safety and liveness. Both are associated with a predicate P: a predicate is a boolean formula that applies to the trace.

Rather than repeating the definitions in the paper, we refer you to the Safety and Liveness definitions on page 2 of the paper.