KTH ROYAL INSTITUTE
OF TECHNOLOGY

Global state

Vladimir Vlassov and Johan Montelius

Global state

Time is very much related to the notion of a ***global state***.

If we cannot agree on a time, how should we agree on a global state?

The global state is important:

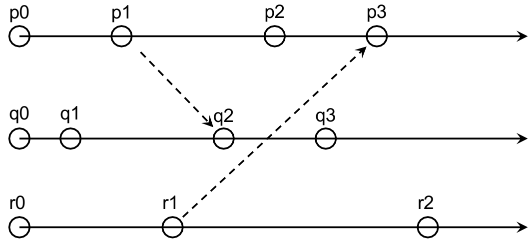
- Garbage collection
- Deadlock detection
- Termination
- Debugging

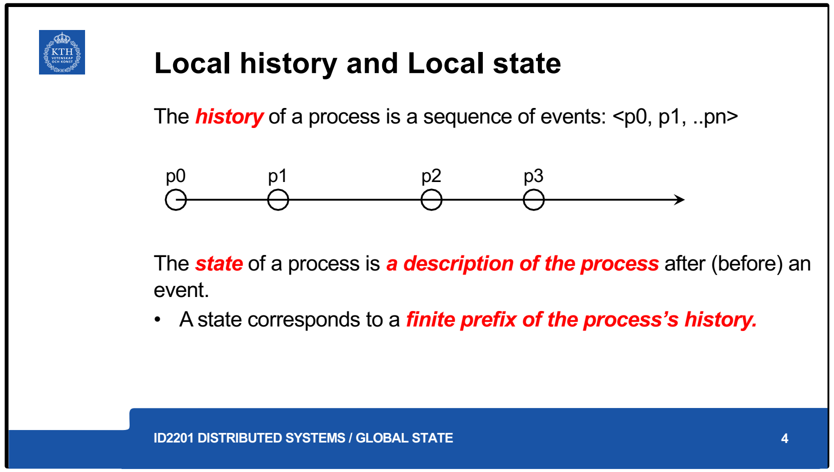
ID2201 DISTRIBUTED SYSTEMS / GLOBAL STATE 2

Distributed garbage collection: An object is a garbage if there are no longer any references to it anywhere in the distributed system. The memory taken up by that object can be reclaimed once it is known to be garbage. To check that an object is garbage, we must verify that there are no references to it anywhere in the system, including outstanding messages in commination channels that may carry a reference to an object that is not referenced by any process. ***Distributed deadlock detection***: A distributed deadlock occurs when each process in a collection of processes waits for another process to send a message and where there is a cycle in the graph of this 'waits-for' relationship. ***Distributed termination detection***: The problem here is how to detect that a distributed algorithm has terminated, i.e., all processing is passive (or idle), and there are no outstanding messages in channels to be received and processed by an idle destination.

Global state

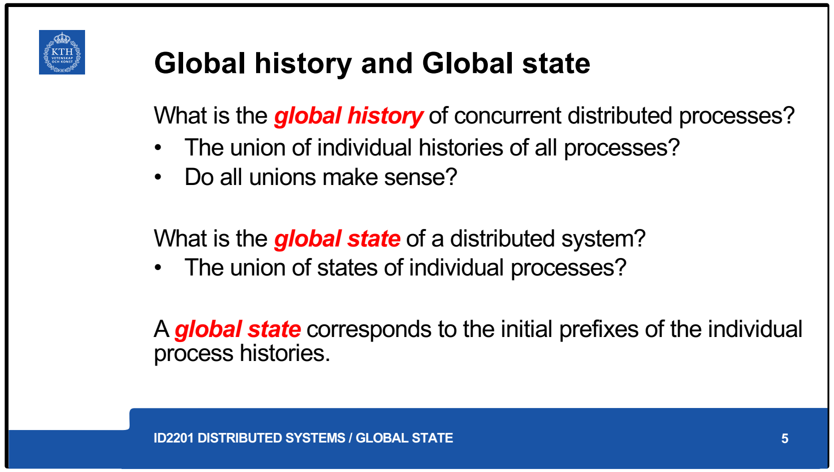
Given a partial order of events, can we say anything about the state of the system?



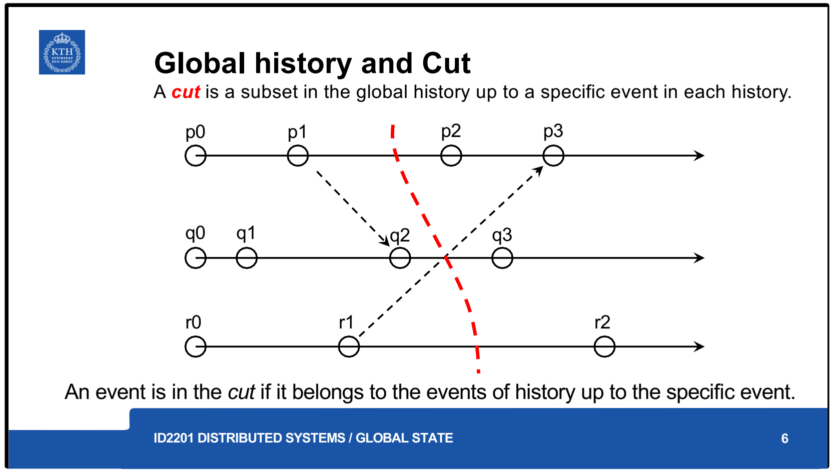


We said that a series of events occur at each process and that we may characterize the execution of each process by its history: as a sequence of all events which causes state transitions of the process. Similarly, we may consider any **finite prefix of the process's history** as a final sequence of events up to some specific event. Each event is either an internal action of the process (for example, updating one of its variables) or the sending or receiving a message over the communication channels that connect the processes.

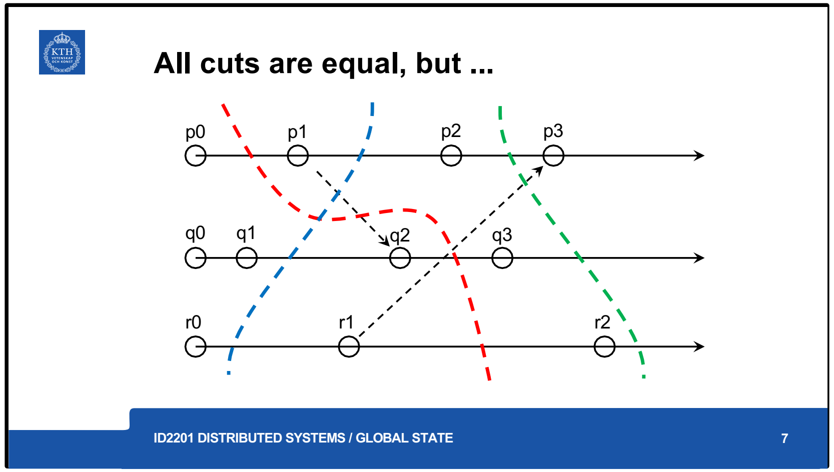
Note that ***the state of the communication channels is sometimes relevant***. Rather than introducing a new type of state, we make the processes record the sending or receiving of all messages as part of their state. If a message was sent but has not been received yet, we can infer whether the message is part of the state of the channel between the sender and the receiver.



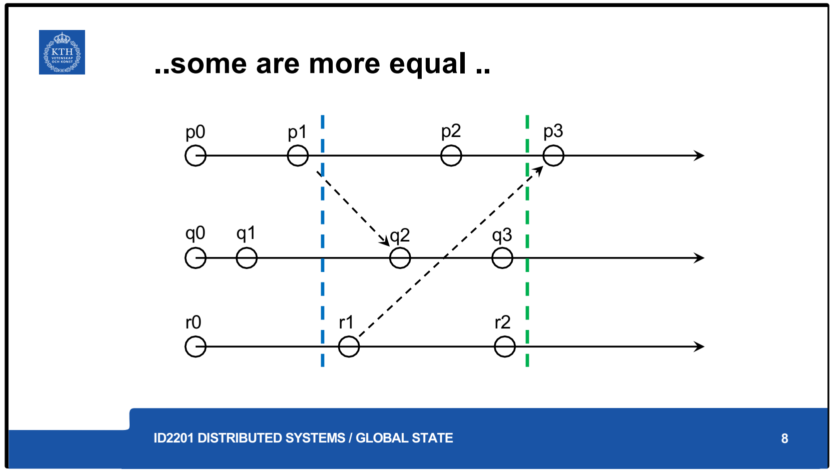
We can also form the ***global history*** of a system of processes as the union of the individual process histories. Mathematically, we can take any set of states of the individual processes to form a ***global state***. But which global states are meaningful – that is, which process states could have **occurred simultaneously**? ***A global state <u>corresponds</u> to the initial prefixes of the individual process histories***.



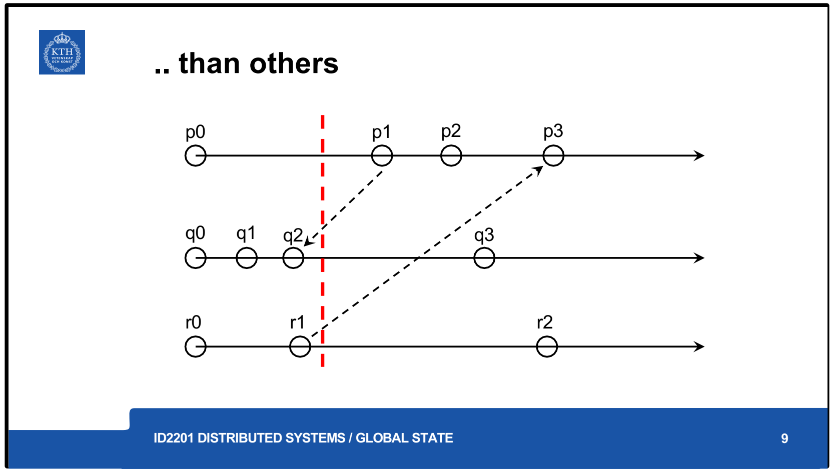
A cut of the system's execution is a subset of its global history that is a union of prefixes of process histories. ***The state of a process in the global state corresponding to a cut*** is a state of the process immediately after the last event processed by the process in the cut. ***The set of the last events*** corresponding to the cut is called the ***frontier*** of the cut.
Note that ***the state of the communication channels is sometimes relevant***. Rather than introducing a new type of state, we make the processes record the sending or receiving of all messages as part of their state. If a message was sent but has not been received yet, we can infer whether the message is part of the state of the channel between the sender and the receiver.



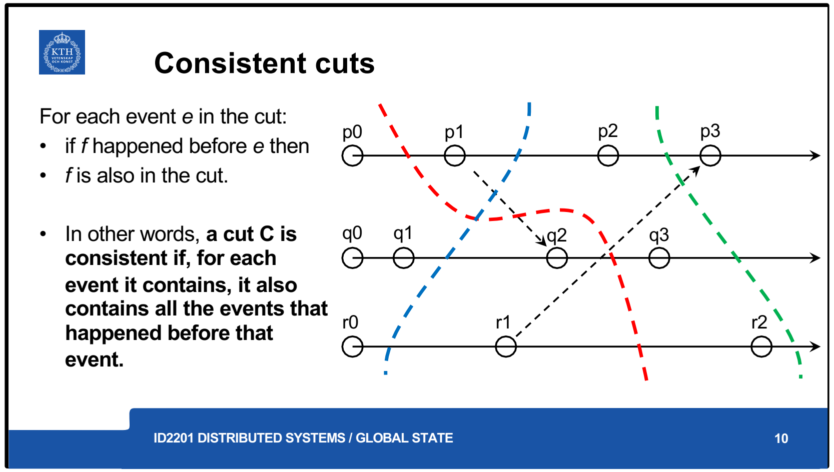
The red cut is inconsistent. This is because, at process *q,* it includes receiving a message from p, but at p, it does not include sending that message. This shows an "effect" (receiving) without a "cause" (sending). The actual execution never was in a global state corresponding to the process states at that frontier (red cut). We can tell this by examining the "happened before" relation between events. ***By contrast, the green cut is consistent***. It includes both the sending and the receiving of a message (p1 -> q2) and the sending but not the receiving of a message from r1. That is consistent with the actual execution – after all, the message took some time to arrive. ***Similarly, the blue cut is also consistent.*** A **cut C is consistent if, for each event it contains, it also contains all the events that happened before that event.**



The green cut is consistent. It includes both the sending and the receiving of a message (p1 -> q2) and the sending but not the receipt of a message from r1. That is consistent with the actual execution – after all, the message took some time to arrive. ***Similarly, the blue cut is also consistent.***



The red cut is inconsistent. This is because, at process *q,* it includes receiving a message from p, but at p, it does not include sending that message. This shows an “effect” (receiving) without a “cause” (sending). The actual execution never was in a global state corresponding to the process states at that frontier (red cut). We can, in principle, tell this by examining the “happened before“ relation between events.



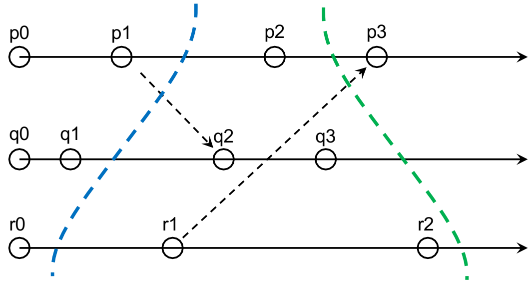
In other words, **a cut C is consistent if, for each event it contains, it also contains all the events that happened before that event.**

Consistent global state

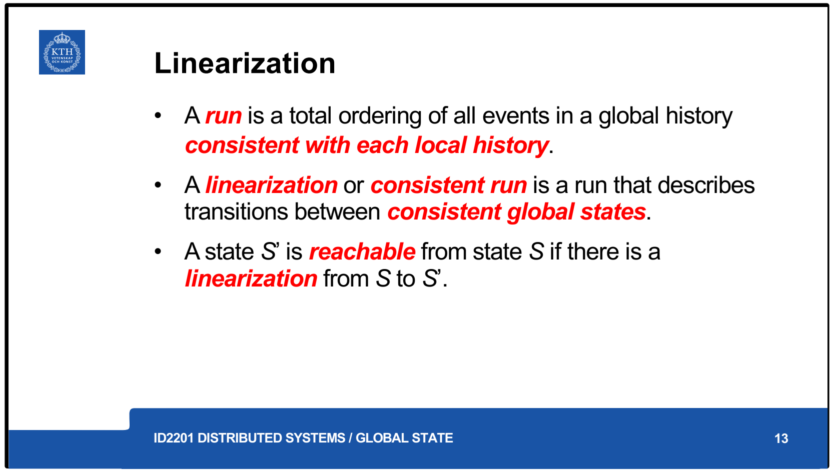
A ***consistent cut*** corresponds to a ***consistent global state***.

- It is a ***possible state*** without contradictions
- it is ***consistent with*** the ***actual execution***
- the actual execution might not have passed through the state, even though it's consistent

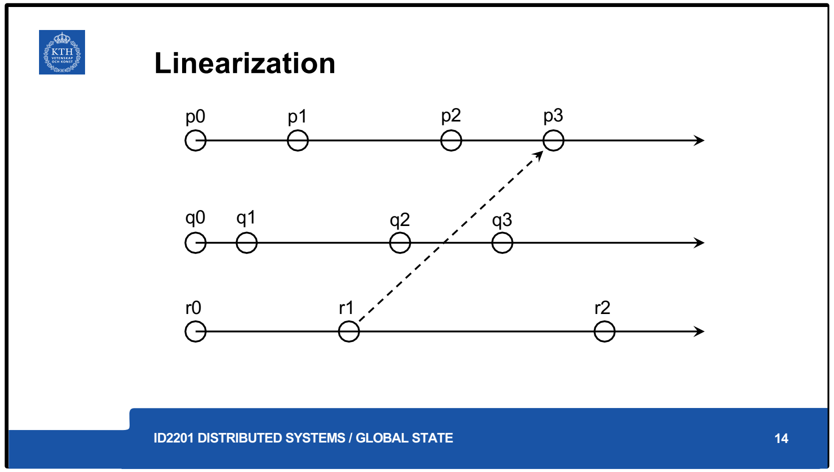
Consistent, but not actual states



All *real-time cuts* are *consistent*, but who knows the real time?



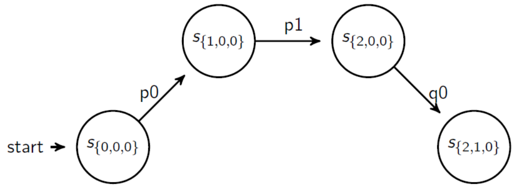
In other words, ***a linearization or consistent run*** is an ordering of the events in global history consistent with this happened-before relation in global history. Note that linearization is also a run. Not all runs pass through consistent global states, but **all linearizations pass only through consistent global states**. Sometimes we may **alter the ordering of concurrent events within a linearization** and derive a run that still passes through only consistent global states. For example, if two successive events in a linearization are the receipt of messages by two processes, then we may swap the order of these two events.



We may characterize **the execution of a distributed system as a series of transitions between global states of the system, i.e., consistent run or linearization**. In each transition, precisely one event occurs at some single process in the system. This event is either the sending of a message, the receiving of a message, or an internal event. If two events happened simultaneously, we might nonetheless deem them to have occurred in a definite order, according to process identifiers. (Events that occur simultaneously must be concurrent: neither happened before the other.) A system evolves in this way through consistent global states.

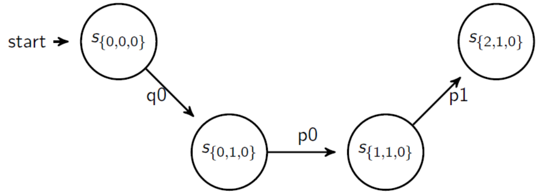
Possible state transitions

[p0, p1, q0, r0, q1, r1, p2, p3, q2, r2, q3]

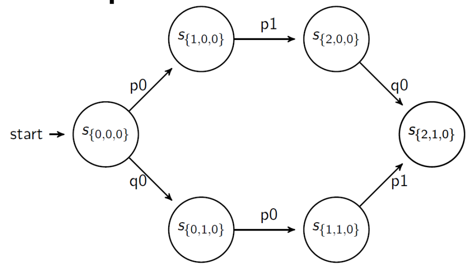


Possible state transitions

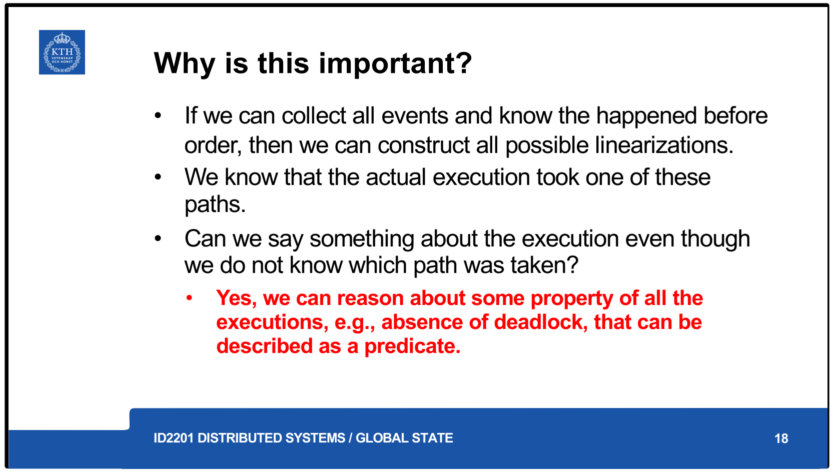
[q0, p0, p1, r0, q1, r1, p2, p3, q2, r2, q3]



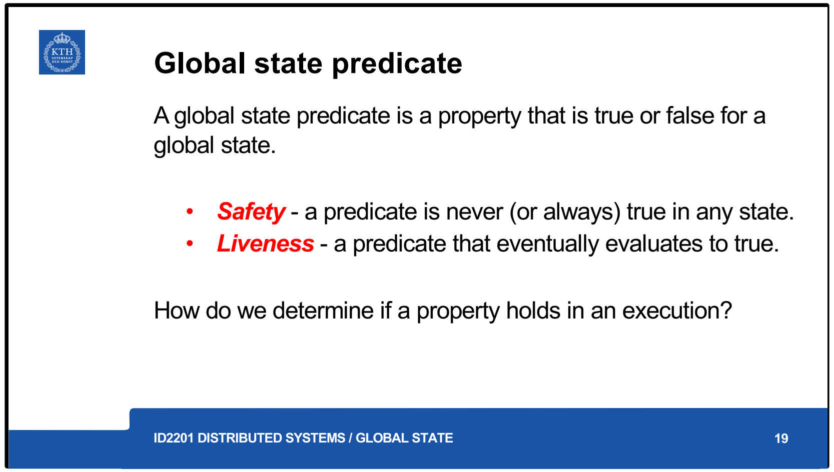
Possible paths



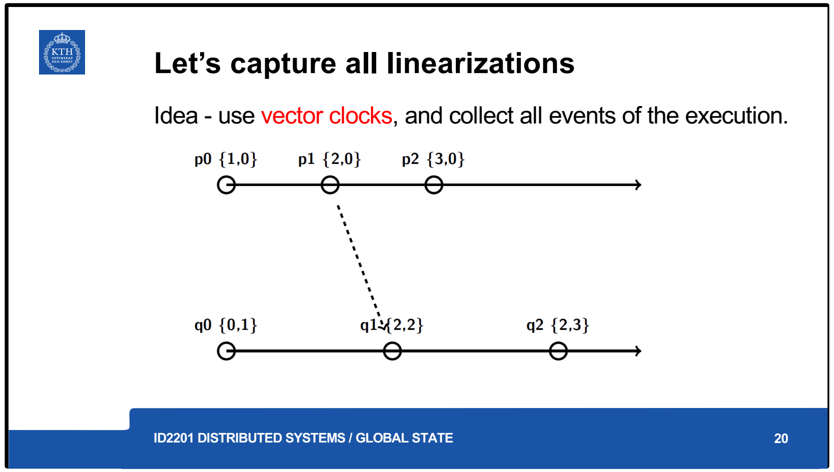
Each path is a consistent run, a linearization, one of which the execution actually took.



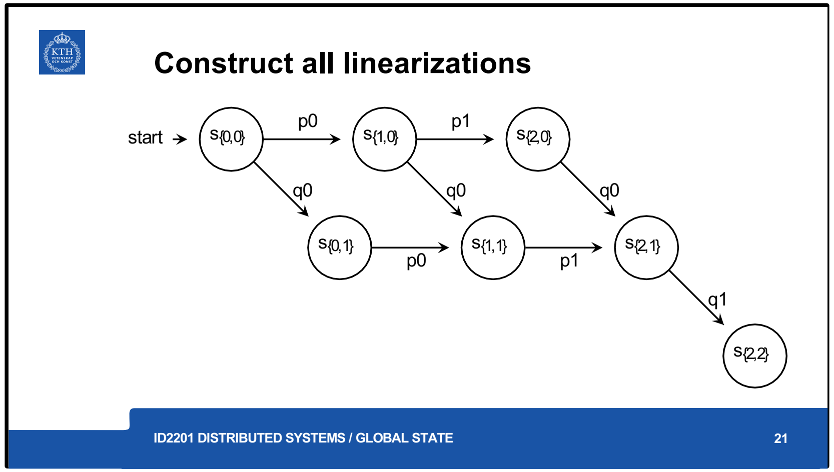
Yes, we can reason about some property of all the executions, e.g., absence of deadlock, that is described as a predicate.



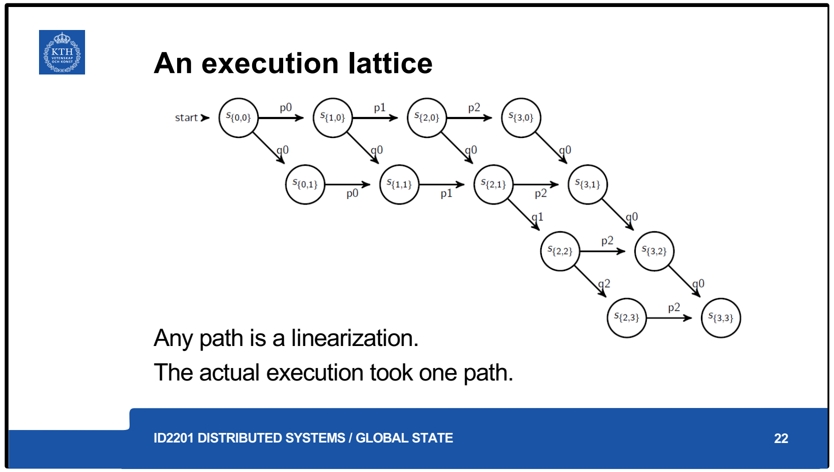
A ***global state predicate*** is a function that maps from the set of global states of processes in the system to {*True*, *False*}. One of the valuable characteristics of the predicates associated with the state of an object being ***garbage***, the system being ***deadlocked,*** or the system being terminated is that they are all ***stable***. Once the system enters a state in which the predicate is *True*, it remains *true* in all future states reachable from that state. By contrast, when we monitor or ***debug*** an application, we are often interested in ***non-stable predicates***, such as that in our example of variables whose difference is supposed to be bounded. Even if the application reaches a state in which the bound obtains, it need not stay in that state.



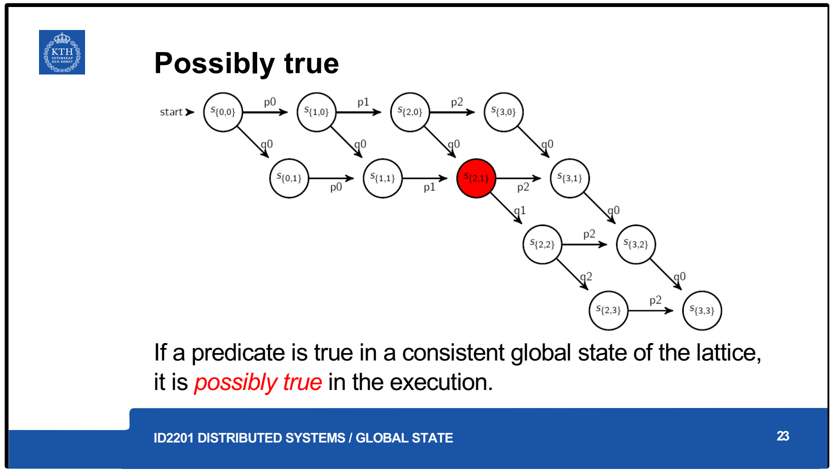
In the text, there is Marzullo and Neiger's algorithm for deriving assertions about whether a predicate held or may have held in the actual run. This algorithm employs a monitor process to collect states. The monitor examines vector timestamps to extract consistent global states and constructs and examines the lattice of all consistent global states. This algorithm involves great computational complexity but is valuable for understanding and can be of some practical benefit in real systems where relatively few events change the global predicate's value. It uses a monitor process that collects all local states of the processes and builds consistent global states from local states time-stamped with a vector clock. Let S= (s1, s2,…, sN) be a global state drawn from the state messages that the monitor has received. Let V (si) be the vector timestamp of the state si received from pi. **Then it can be shown that S is a consistent global state if and only if: V(si)[i] ≥ V(sj)[i] for i, j = 1, 2, … N – (Condition CGS), i.e., i-th element in the vector that came from pi is at least (more or equal) as i-th in other vector clocks.**



The monitor process construct runs (linearization) from the reporting events with vector-clock time stamps;
The lattice of consistent global states is a structure (graph) that captures the relation of reachability between consistent global states. The nodes denote global states, and the edges denote possible transitions between these states.

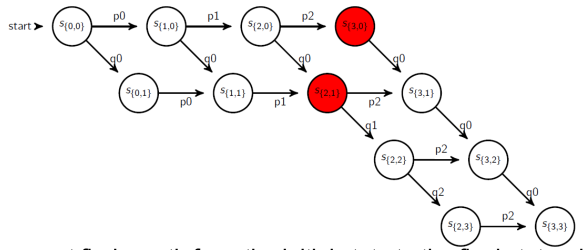


The lattice of consistent global states is a structure (graph) that captures the relation of reachability between consistent global states. The nodes denote global states, and the edges denote possible transitions between these states.

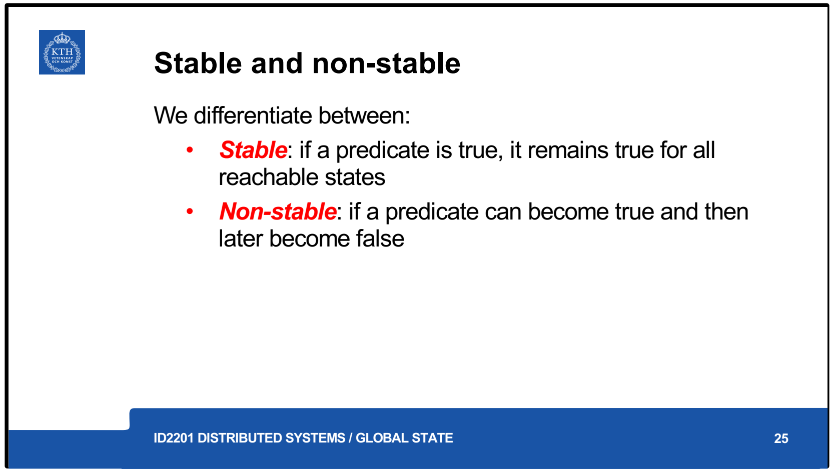


Assuming that it’s stable, It is possibly true that the actual execution has passed through that state.

Definitely true



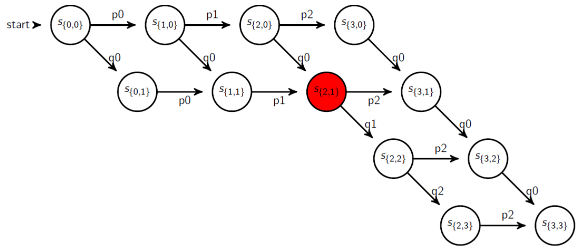
If we cannot find a path from the initial state to the final state without reaching a state for which a predicate is true, then the predicate is *definitely true* during the execution.



A ***global state predicate*** is a function that maps from the set of global states of processes in the system to {*True*, *False*}. One of the valuable characteristics of the predicates associated with the state of an object being ***garbage***, the system being ***deadlocked,*** or the system being terminated is that they are all ***stable***. Once the system enters a state in which the predicate is *True*, it remains *true* in all future states reachable from that state. By contrast, when we monitor or ***debug*** an application, we are often interested in ***non-stable predicates***, such as that in our example of variables whose difference is supposed to be bounded. Even if the application reaches a state in which the bound obtains, it need not stay in that state.

Stable is good

What do I know if a stable predicate is true for state $S_{\{2,1\}}$?



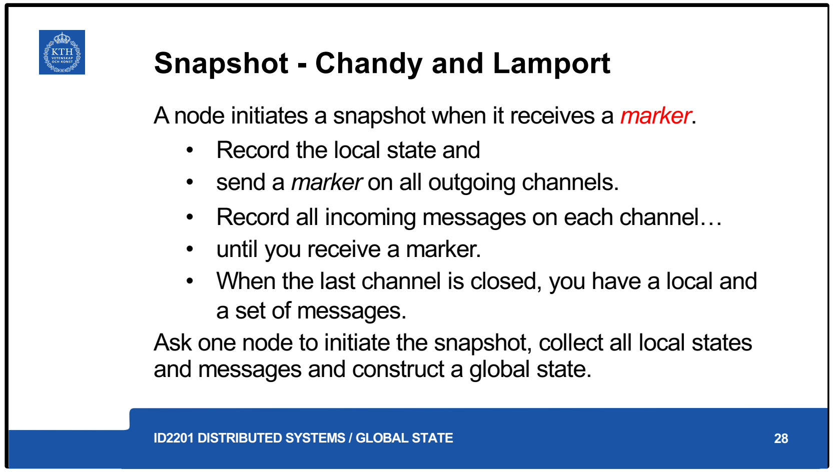
Let’s capture a possible state

Idea: capture a consistent global state that was possibly true in the execution.

If a stable predicate is true for this state, it is true in the actual execution.

How do we capture a state?

Capture – get it!



An obvious method for gathering the state is for all processes to send the local states they recorded to a designated collector process.

The Chandy and Lamport snapshot algorithm assumes that:

• Neither channels nor processes fail – communication is reliable so that every message sent is eventually received intact, exactly once.

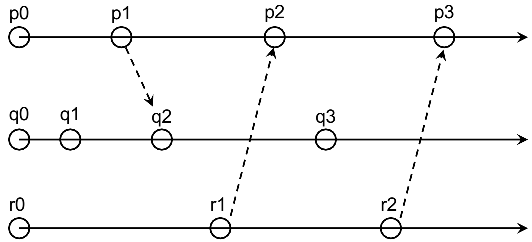
• Channels are unidirectional and provide FIFO-ordered message delivery.

• The graph of processes and channels is strongly connected (there is a path between any two processes).

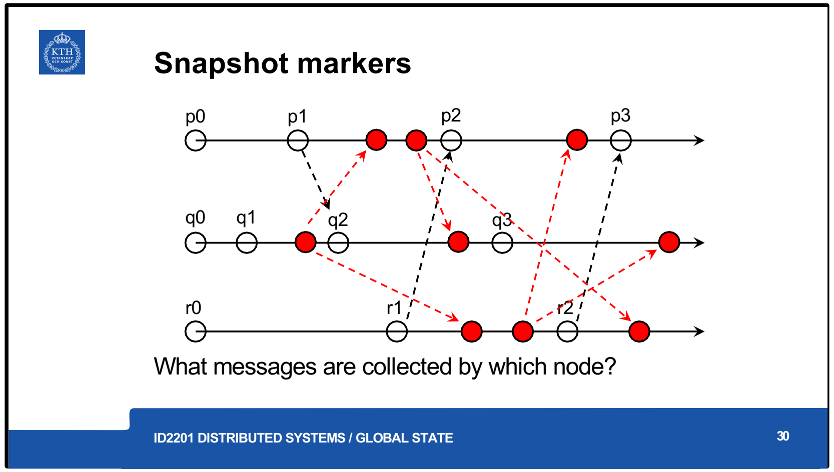
• Any process may initiate a global snapshot at any time.

• The processes may continue executing and send and receive regular messages while the snapshot takes place.

Snapshot markers



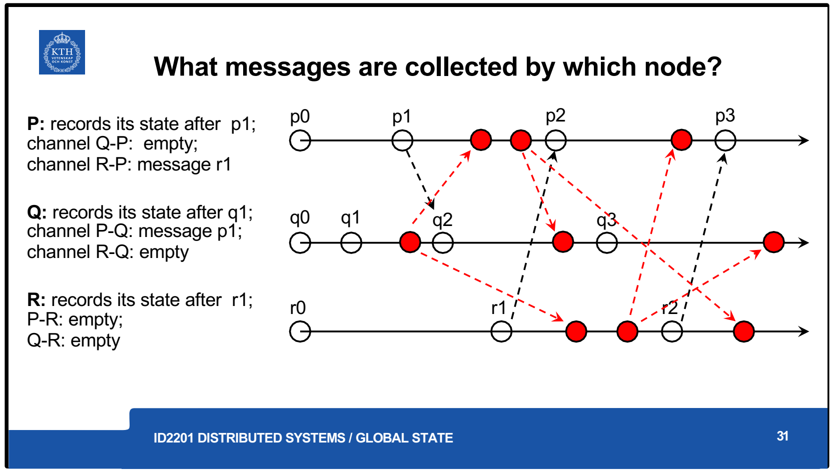
What messages are collected by which node?



Q: records its state after q1; state of the channel P-Q: message p1; R-Q: empty

P: records its state after p1; Q-P: empty; R-P: message r1

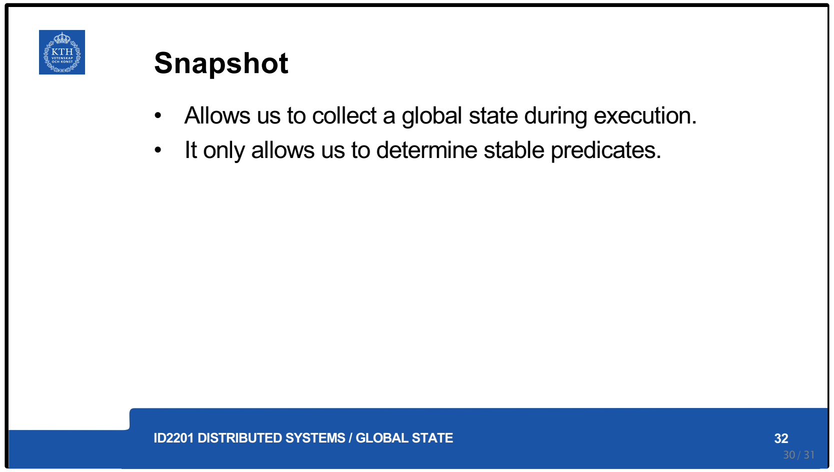
R: records its state after r1; P-R: empty; Q-R: empty



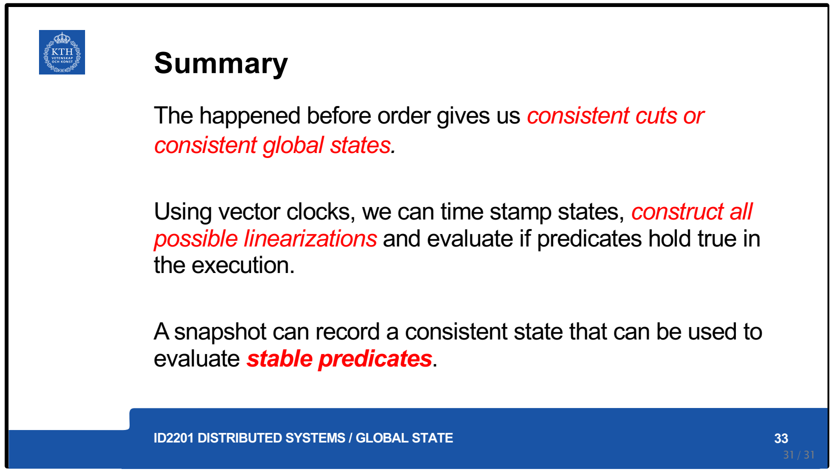
Q: records its state after q1; state of the channel P-Q: message p1; R-Q: empty

P: records its state after p1; Q-P: empty; R-P: message r1

R: records its state after r1; P-R: empty; Q-R: empty



Why only global?



We introduced the ***concepts of events, local and global histories, cuts, local and global states, runs, consistent states, linearizations (consistent runs), and reachability***. A consistent state or run is one that is in accord with the happened-before relation. We went on to consider ***the problem of recording a consistent global state*** by observing a system's execution. Our ***objective was to evaluate a predicate on this state***. An important class of predicates is the stable predicates. We described ***the snapshot algorithm of Chandy and Lamport***, which captures a consistent global state and allows us to make assertions about whether a stable predicate holds in the actual execution. In the text, there is ***Marzullo and Neiger's algorithm*** for deriving assertions about whether a predicate held or may have held in the actual run. ***This algorithm employs a monitor process to collect states***. The monitor examines vector timestamps to extract consistent global states and ***constructs and examines the lattice of all consistent global states***. This algorithm involves great computational complexity but is valuable for understanding and can be of some practical benefit in real systems where relatively few events change the global predicate's value. The algorithm has a more efficient variant in synchronous systems, where clocks may be synchronized.