Distributed Algorithms

LECTURER: V.A. ZAKHAROV

Lecture 11.

Snapshot problem
Chandy–Lamport algorithm
Lai–Yang algorithm
Applications of snapshot algorithms
Deadlock detection

Snapshot problem

Self-control is peculiar to human being.

And there are also cases when a distributed system needs to «look» at its own behavior in order to assess whether everything is in order.

It is not easy to observe the computation of a distributed system from **within** the same system.

An important building block in the design of algorithms operating on system computations is a procedure for computing and storing a single configuration of this computation, a so-called **snapshot** .

Snapshot problem

An execution of a distributed system is a sequence of configurations. A configuration of a distributed system is a set $\gamma = (c_{p_1}, c_{p_2}, \ldots, c_{p_N}, M)$ of process states $c_{p_1}, c_{p_2}, \ldots, c_{p_N}$ plus the content M of the channels. Every process p_i can store its current state c_{p_i} . However, not every set of process states forms a system configuration.

How to make distributed system processes to remember their states **jointly, in concert**?

Snapshot problem

The construction of snapshots is motivated by several applications.

1). Properties of the computation can be analyzed **off-line** by an algorithm that inspects the (fixed) snapshot rather than the (varying) actual process states. These properties include **stable properties**.

A property P of configurations is stable if

$$P(\gamma) \wedge \gamma \rightsquigarrow \delta \implies P(\delta),$$

i.e, if a computation ever reaches a configuration γ for which P holds, P remains true in every configuration δ from then on.

Examples of stable properties include termination, deadlock, loss of tokens, and non-reachability of objects in dynamic memory structures.

Snapshot problem

2) A snapshot can be used instead of the initial configuration if the computation must be restarted due to a process failure.

To this end, the local state c_p for process p , captured in the snapshot, is restored in that process, after which the operation of the algorithm is continued.

Snapshot problem

2) A snapshot can be used instead of the initial configuration if the computation must be restarted due to a process failure.

To this end, the local state c_p for process p , captured in the snapshot, is restored in that process, after which the operation of the algorithm is continued.

3) Snapshots are a useful tool in debugging distributed programs. An off-line analysis of a configuration taken from an erroneous execution may reveal why a program does not act as expected.

Snapshot problem

Consider a computation C of a distributed system, consisting of a set of processes $\mathbb{P}$.

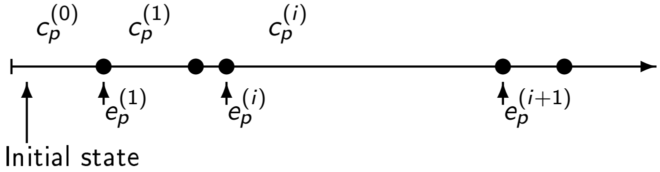
We make the weak fairness assumption that every message will be received in finite time, and it is assumed that the network is (strongly) connected.

Denote by Ev the set of events of this computation.

The local computation of a process p is a sequence $c_p^{(0)}, c_p^{(1)}, \ldots$ of process states, where $c_p^{(0)}$ is an initial state of p .
The transition from $c_p^{(i-1)}$ to $c_p^{(i)}$ is caused by the occurrence of an event $e_p^{(i)}$ in p .

Thus, $Ev = \bigcup_{p \in \mathbb{P}} \{e_p^{(1)}, e_p^{(2)}, \ldots\}$.

Snapshot problem



Snapshot problem

On the set of events of a process p a causal order $\preceq_p$ is defined as follows:

$$e_p^{(i)} \preceq_p e_p^{(j)} \iff i \leq j.$$

Every event is either a send event , a receive event , or an internal event .

To simplify the representation of the algorithms and theorems it will be assumed that the entire communication history of a process is reflected in its state .

This means that for any channel from p to q a state $c_p^{(i)}$ of a process p includes the list $sent_{pq}^{(i)}$ of all messages sent from p to q in the events $e_p^{(1)}$ through $e_p^{(i)}$, i.e. every process keeps track of all messages it sent to its neighbors. The process q also keeps track of all messages received from p in the list $rcvd_{pq}^{(i)}$.

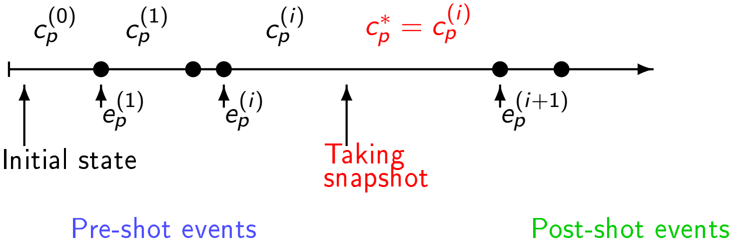
Snapshot problem

The aim of a snapshot algorithm is to construct explicitly a system configuration composed from local states (snapshot states) of each process.

A process p takes a snapshot of its local state by storing a local state c_p^* which is called a local snapshot of p .

If $c_p^{(i)}$ is a local snapshot of a process, i.e. p takes this snapshot between the events $e_p^{(i)}$ and $e_p^{(i+1)}$, then the events $e_p^{(j)}$ such that $j \leq i$ are called pre-shot events of p , and the events with $j > i$ are called post-shot events of p .

Snapshot problem



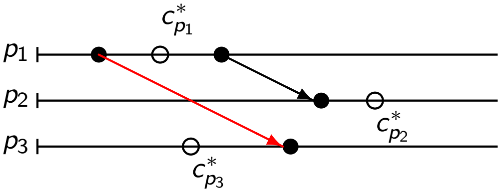
Snapshot problem

Global snapshot S^* is formed of the local snapshots c_p^* for all processes p in $\mathbb{P}$. It will be denoted as $S^* = (c_{p_1}^*, \ldots, c_{p_N}^*)$.

Because local states include communication histories, a snapshot S^* defines a configuration γ^* ; the state of a channel pq is defined to be the set of messages sent by p (according to c_p^*), but not received by q (according to c_q^*).
In other words, the state of a channel pq in a global snapshot S^* is defined as the list $sent_{pq}^* \setminus rcvd_{pq}^*$.

The configuration consisting of the snapshot states and the defined channel states will be denoted γ^* .

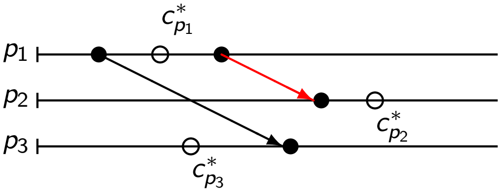
Snapshot problem



Some anomalies in the construction of the configuration γ^* arise if $rcvd^*_{pq}$ is not a subset of $sent^*_{pq}$. According to state $c^*_{p_1}$ in the collected snapshot a message was sent from p_1 to p_3 , but, according to the local state $c^*_{p_3}$, no message was received from p_1 .

Thus, the channel $p_1 p_3$ contains one message in the snapshot, and this message is said to be «in transit» in the snapshot.

Snapshot problem



Look at the message which p_1 sent to p_2 .

The send event for this message is a post-shot event, whereas the receiving of this message is a pre-shot event.

Thus, according to the state $c^*_{p_1}$ no messages were sent via the channel $p_1 p_2$, while at the state $c^*_{p_2}$ it is noted that some message was received via this channel. Since $rcvd^*_{p_1 p_2} \not\subseteq sent^*_{p_1 p_2}$, no meaningful choice for the state of channel $p_1 p_2$ can be made.

Snapshot problem

Definition 1.
A snapshot S^* is called feasible , if $rcvd^*_{pq} \subseteq sent^*_{pq}$ holds for every pair of neighboring processes p и q .

The feasibility of a snapshot implies that in the construction of the implied configuration γ no messages «remain» in $rcvd^*_{pq}$ that are not stored in $sent^*_{pq}$.

We shall call a message a pre-shot message (or post-shot message , respectively) if it is sent in a pre-shot event (or post-shot event, respectively).

Snapshot problem

There is a one-to-one correspondence between snapshots and finite **cuts** in the event collection of the computation. A cut is a collection of events that is left-closed with respect to local causality.

Definition 2.

A cut on the set Ev is any subset $L \subseteq Ev$ which complies with the following requirement

$$e \in L \wedge e' \preceq_p e \quad \Longrightarrow \quad e' \in L.$$

A cut L_2 is said to be **late than** a cut L_1 iff $L_1 \subseteq L_2$.

A consistent cut on the set of events Ev is such a cut L which satisfies

$$e \in L \wedge e' \preceq e \quad \Longrightarrow \quad e' \in L.$$

Snapshot problem

Simple questions: check yourself.

1. Give an example of such a cut on the set of events which is not consistent.

2. Is it true that a set of all pre-shot events for some snapshot γ^* is a feasible cut ?

Snapshot problem

It is easy to see that for every global snapshot S^* the set L of all pre-shot events is a finite cut. We will say that such a cut L is induced by S^* .

Consider now an arbitrary cut L .

For every process p either no event in p is included in L (in this case we will assume $m_p = 0$), or L includes a maximal event $e_p^{(m_p)}$ such that all events $e \preceq_p e_p^{(m_p)}$ are also included in L .

Therefore, L is exactly the set of pre-shot events of the snapshot defined by $S^* = (c_{p_1}^{(m_{p_1})}, \ldots, c_{p_N}^{(m_{p_N})})$.

Snapshot problem

A snapshot will be used to derive information about the computation from which it is taken, but an arbitrarily taken snapshot provides little information about this computation.
we would like the snapshot algorithm to compute a configuration that «actually occurs» in the computation.

However the set of configurations which occur in an arbitrary execution of a system is not uniquely defined by a computation of a system.
Thus we shall accept any configuration that is possible for the computation (i.e., occurs in some execution of the computation) as a meaningful output of the algorithm.

Задача сохранения моментального состояния

Definition 3.
A snapshot S^* is called meaningful in a computation C , if ther exists such an execution $E \in C$ that γ^* is a configuration in E .

We require the snapshot algorithm to coordinate the registration of the local snapshots in such a way that the resulting global snapshot is meaningful.

Snapshot problem

CHECK YOURSELF QUESTIONS

1. What is a difference between a cut and a consistent cut? Find an example of a cut which is not consistent.

2. Let $Clock(p, e)$ be a Lamport clock for the process p . Is a subset of events of an execution

$$L_k = \bigcup_{p \in \mathbb{P}} \{e : Clock(p, e) \leq k\}$$

- is a cut on the set of events Ev ?
- is a consistent cut on the set of events Ev ?

Snapshot problem

Theorem 1.
Let S^* be a snapshot of a system and L be a cut induced by S^* . The following three statements are equivalent:

1) S^* is feasible;
2) L is consistent;
3) S^* is meaningful.

Snapshot problem

Theorem 1.

Let S^* be a snapshot of a system and L be a cut induced by S^* . The following three statements are equivalent:

1) S^* is feasible;
2) L is consistent;
3) S^* is meaningful.

Proof.

We show that

$$(1) \implies (2) \implies (3) \implies (1)$$

Snapshot problem

Proof. (1) $\Rightarrow$ (2)

Assume that the cut S^* is feasible. To show that L is consistent take arbitrary $e \in L$ and $e' \preceq e$. By the definition of $\preceq$ it is enough to prove that $e' \in L$ holds in two following cases.

Snapshot problem

Proof. (1) $\Rightarrow$ (2)

Assume that the cut S^* is feasible. To show that L is consistent take arbitrary $e \in L$ and $e' \preceq e$. By the definition of $\preceq$ it is enough to prove that $e' \in L$ holds in two following cases.

1. $e' \preceq_p e$, where p is a process e' and e take place. In this case $e' \in L$ holds since L is a cut.

Snapshot problem

Proof. (1) $\Rightarrow$ (2)

Assume that the cut S^* is feasible. To show that L is consistent take arbitrary $e \in L$ and $e' \preceq e$. By the definition of $\preceq$ it is enough to prove that $e' \in L$ holds in two following cases.

1. $e' \preceq_p e$, where p is a process e' and e take place. In this case $e' \in L$ holds since L is a cut.

2. e' is a send event, and e is the corresponding receive event. Consider a process p , where e' takes place, and a process q , where e takes place, and let m be a message exchanged in these events. Then

$$
\begin{aligned}
e \in L \quad & \Longrightarrow \quad m \in rcvd^*_{pq}, && \text{since } e \text{ is a pre-shot event} \\
& \Longrightarrow \quad m \in sent^*_{pq}, && \text{since } S^* \text{ is feasible} \\
& \Longrightarrow \quad e' \in L.
\end{aligned}
$$

Snapshot problem

Proof. (2) $\Rightarrow$ (3)

Snapshot problem

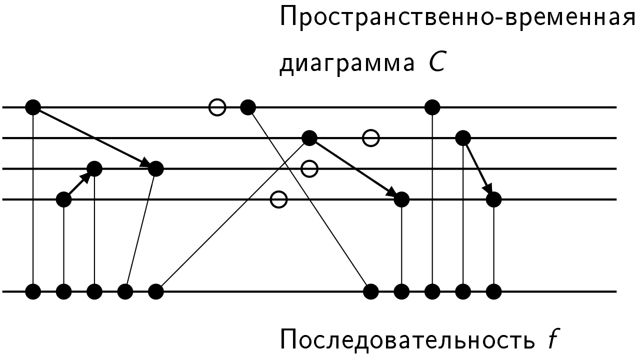
Proof. (2) $\Rightarrow$ (3)

Construct an execution in which all pre-shot events occur before all post-shot events.

Let $f = (f_0, f_1, \ldots)$ be an enumeration of the set of events Ev defined as follows.

First f lists all pre-shot events in Ev in any order consistent with the causal relation $\preceq$, and then lists all post-shot events in any order consistent with $\preceq$.

Snapshot problem



Snapshot problem

Proof. (2) $\Rightarrow$ (3)

Snapshot problem

Proof. (2) $\Rightarrow$ (3)

Construct an execution in which all pre-shot events occur before all post-shot events.

Let $f = (f_0, f_1, \ldots)$ be an enumeration of the set of events Ev defined as follows.

First f lists all pre-shot events in Ev in any order consistent with the causal relation $\preceq$, and then lists all post-shot events in any order consistent with $\preceq$.

Apply then Theorem 2 (see Lecture 2) about the executions of distributed systems.

Let $f = (f_0, f_1, f_2, \ldots)$ be a permutation of events of an execution E , which preserves the causal order of events. Then f defines the unique execution F , which begins with the same configuration as the execution E .

Snapshot problem

In order to apply this theorem it must be shown that the entire sequence f is consistent with $\preceq$. Let $f_i \preceq f_j$.

Snapshot problem

In order to apply this theorem it must be shown that the entire sequence f is consistent with $\preceq$. Let $f_i \preceq f_j$.

If both events f_i and f_j are pre-shot, then $i \leq j$, since in f all preshot events follow in the order consistent with $\preceq$.

Snapshot problem

In order to apply this theorem it must be shown that the entire sequence f is consistent with $\preceq$. Let $f_i \preceq f_j$.

If both events f_i and f_j are pre-shot, then $i \leq j$, since in f all preshot events follow in the order consistent with $\preceq$.

By the same reason the inequality holds whenboth f_i and f_j are post-shot events.

Snapshot problem

In order to apply this theorem it must be shown that the entire sequence f is consistent with $\preceq$. Let $f_i \preceq f_j$.

If both events f_i and f_j are pre-shot, then $i \leq j$, since in f all preshot events follow in the order consistent with $\preceq$.

By the same reason the inequality holds whenboth f_i and f_j are post-shot events.

If f_i is a pre-shot event, and f_j is a post-shot event then $i \leq j$ holds, since in f all pre-shot events precede all post-shot events.

Snapshot problem

In order to apply this theorem it must be shown that the entire sequence f is consistent with $\preceq$. Let $f_i \preceq f_j$.

If both events f_i and f_j are pre-shot, then $i \leq j$, since in f all preshot events follow in the order consistent with $\preceq$.

By the same reason the inequality holds whenboth f_i and f_j are post-shot events.

If f_i is a pre-shot event, and f_j is a post-shot event then $i \leq j$ holds, since in f all pre-shot events precede all post-shot events.

The case when f_j is a pre-shot event, and f_i is a post-shot event is impossible, because L is a consistent cut.

Snapshot problem

In order to apply this theorem it must be shown that the entire sequence f is consistent with $\preceq$. Let $f_i \preceq f_j$.

If both events f_i and f_j are pre-shot, then $i \leq j$, since in f all preshot events follow in the order consistent with $\preceq$.

By the same reason the inequality holds whenboth f_i and f_j are post-shot events.

If f_i is a pre-shot event, and f_j is a post-shot event then $i \leq j$ holds, since in f all pre-shot events precede all post-shot events.

The case when f_j is a pre-shot event, and f_i is a post-shot event is impossible, because L is a consistent cut.

Hence, f preserves the causal order $\preceq$. Therefore, by Theorem on executions there exists such an execution F , which consists of all events from Ev , which occur in the same order as defined by the sequence f . The execution F contains a configuration γ^* immediately after the execution of all pre-shot events..

Snapshot problem

Proof. (3) $\Rightarrow$ (1)

If the snapshot S^* is meaningful, then a configuration γ^* occurs in an execution of C .

In each execution a message is sent before it is received.

This implies that $rcvd^*_{pq} \subseteq sent^*_{pq}$ holds for every pair of processes p and q .

Hence, the snapshot S^* is feasible. $\square$

Chandy–Lamport Algorithm

By Theorem 1, it suffices to coordinate the local snapshots so as to guarantee that the resulting snapshot is feasible. This simplifies the requirements of the snapshot algorithm to the following two properties.

1. The taking of a local snapshot must be triggered in each process.
2. No post-shot message is received in a pre-shot event.

Chandy–Lamport Algorithm

By Theorem 1, it suffices to coordinate the local snapshots so as to guarantee that the resulting snapshot is feasible. This simplifies the requirements of the snapshot algorithm to the following two properties.

1. The taking of a local snapshot must be triggered in each process.
2. No post-shot message is received in a pre-shot event.

In all snapshot algorithms it is ensured that a process takes its snapshot before the receipt of a post-shot message.

To distinguish the messages of the snapshot algorithm from the messages of the computation proper, the former are called control messsages and the latter are called basic messages .

Chandy–Lamport Algorithm

This algorithm is applicable to any strongly connected network under the assumption that channels are fifo, i.e., messages sent via any single channel are received in the same order as they were sent.

Chandy–Lamport Algorithm

This algorithm is applicable to any strongly connected network under the assumption that channels are fifo, i.e., messages sent via any single channel are received in the same order as they were sent.

The sketch of the Algorithm.

In the Chandy–Lamport Algorithm processes inform each other about the snapshot construction by sending special messages (markers) $\langle \mathbf{mkr} \rangle$ via each channel.

Each process sends markers exactly once, via each adjacent channel, when the process takes its local snapshot; the markers are control messages.

The receipt of a $\langle \mathbf{mkr} \rangle$ message by a process that has not yet taken its snapshot causes this process to take a snapshot and send $\langle \mathbf{mkr} \rangle$ messages as well.

The Algorithm is executed concurrently with the computation C.

Chandy–Lamport Algorithm

var $taken_p$: **bool** **init** $false$;

To initiate the Algorithm:

begin record the local state ; $taken_p := true$;
  forall $q \in Neigh_p$ **do** send $\langle \mathbf{mkr} \rangle$ to q
end

If a marker has arrived:

begin receive $\langle \mathbf{mkr} \rangle$;
  if not $taken_p$ **then**
   begin record the local state ; $taken_p := true$;
     forall $q \in Neigh_p$ **do** send $\langle \mathbf{mkr} \rangle$ to q
   end
end

Chandy–Lamport Algorithm

Lemma.
If at least one process initiates the algorithm, all processes take a local snapshot within finite time.

Chandy–Lamport Algorithm

Lemma.
If at least one process initiates the algorithm, all processes take a local snapshot within finite time.

Proof.
Help yourself; this is easy.

Chandy–Lamport Algorithm

Theorem 2.

The Chandy–Lamport Algorithm computes a meaningful snapshot within finite time after its initialization by at least one process.

Chandy–Lamport Algorithm

Theorem 2.

The Chandy–Lamport Algorithm computes a meaningful snapshot within finite time after its initialization by at least one process.

Proof.

By the Lemma, the algorithm computes a snapshot in finite time. It remains to show that the resulting snapshot is feasible, i.e., that each post-shot (basic) message is received in a post-shot event.

Chandy–Lamport Algorithm

Theorem 2.

The Chandy–Lamport Algorithm computes a meaningful snapshot within finite time after its initialization by at least one process.

Proof.

By the Lemma, the algorithm computes a snapshot in finite time. It remains to show that the resulting snapshot is feasible, i.e., that each post-shot (basic) message is received in a post-shot event.

Consider a post-shot message m , sent by a process p to a process q . Before sending m the process p has taken a local snapshot and sent the message $\langle \mathbf{mkr} \rangle$ to all its neighbors including q .

Because the channels are fifo, the process q received $\langle \mathbf{mkr} \rangle$ before it received m , and, therefore, it took its local snapshot upon receipt of this message or earlier.

Hence, the receipt of m is a post-shot event. $\square$

Chandy–Lamport Algorithm

CHECK YOURSELF QUESTIONS

1. What is the message exchange complexity of the Chandy–Lamport Algorithm?

2. How much sensitive is the Chandy–Lamport Algorithm to a) message losses, b) message duplications, c) message shuffling?

Lai–Yang Algorithm

The algorithm of Lai and Yang does not rely on the fifo property of channels.
Therefore, it is not possible to «separate» pre-shot and post-shot messages by markers as is done in the algorithm of Chandy and Lamport.

Lai–Yang Algorithm

The algorithm of Lai and Yang does not rely on the fifo property of channels.
Therefore, it is not possible to «separate» pre-shot and post-shot messages by markers as is done in the algorithm of Chandy and Lamport.

Sketch of the Algorithm.

Each individual basic message is tagged with information revealing whether it is pre-shot or post-shot

To this end a process p when sending a message in a computation C appends the Boolean value of $taken_p$ to it. Because the contents of the messages of C are not of concern here, we denote these messages simply as $\langle \mathbf{mes}, c \rangle$, where c is a value appended to this message by a sender.

The snapshot algorithm inspects incoming messages and records the local state as it is before receipt of the first postshot message.

Lai–Yang Algorithm

var $taken_p$: **bool** **init** $false$;

To initiate the algorithm:
begin record the local state; $taken_p := true$ **end**

To send a message of C :
send $\langle \mathbf{mes}, taken_p \rangle$

If a message $\langle \mathbf{mes}, c \rangle$ arrives ;
begin receive $\langle \mathbf{mes}, c \rangle$;
    if c **and not** $taken_p$ **then**
      begin record the local state; $taken := true$ **end**;
    change state as in the receive event of C
end

Lai–Yang Algorithm

The Lai–Yang Algorithm exchanges no control messages, but it does not ensure that each process eventually records its state, which it may indeed fail to do. Consider a process p , which is not an initiator of the snapshot algorithm, and assume that the neighbors of p do not send messages to p after taking their local snapshots. In this situation p never records its state, and the snapshot algorithm terminates with an incomplete snapshot.

The solution to this problem depends on what is known about the computation C ; if eventual communication with every process is guaranteed, a complete snapshot will always be taken. Otherwise, the algorithm may be augmented with the initial exchange of control messages between all processes,

Lai–Yang Algorithm

Theorem 3.

The Lai–Yang Algorithm only computes meaningful snapshots.

Lai–Yang Algorithm

Theorem 3.

The Lai–Yang Algorithm only computes meaningful snapshots.

Proof.

Consider a snapshot computed by the Algorithm and let $m = \langle \mathbf{mes}, c \rangle$ be a post-shot message sent by a process p to a process q. This means that $c = \mathit{true}$, and, therefore, q takes its snapshot at the latest upon receipt of m.

Thus, the local snapshot as it was taken by q, does not account the receipt of m, and the event of receiving m is regarded as a post-shot event.

(Recall that it only matters **which** local state is recorded, not **when** it is recorded; in this case, recording may take place simultaneously with the first post-shot event.) $\square$

HOMETASK

1. Give a full description of the Lai–Yang algorithm, including mechanisms to enforce completion of the snapshot and construction of the channel states.

2. Prove the correctness of Your extension of Lai–Yang Algorithm

The applications of snapshots

Consider some stable properties P of configurations, i.e. once an execution reaches a configuration in which P holds, P thereafter holds forever.

The applications of snapshots

Consider some stable properties P of configurations, i.e. once an execution reaches a configuration in which P holds, P thereafter holds forever.

1. **Termination of computation.** If γ is a terminal configuration and $\gamma \rightsquigarrow \delta$, then $\gamma = \delta$, and, therefore, δ is also terminal. Consequently, the termination-detection problem may be solved by computing a snapshot and inspecting it for active processes and basic messages in this snapshot.

The applications of snapshots

Consider some stable properties P of configurations, i.e. once an execution reaches a configuration in which P holds, P thereafter holds forever.

1. **Termination of computation.** If γ is a terminal configuration and $\gamma \rightsquigarrow \delta$, then $\gamma = \delta$, and, therefore, δ is also terminal. Consequently, the termination-detection problem may be solved by computing a snapshot and inspecting it for active processes and basic messages in this snapshot.
2. **Deadlock detection.** If in configuration γ a subset S of processes is blocked because all processes in S are waiting for other processes in S , the same holds in later configurations, even though processes outside S may change their states. Therefore, snapshots may be helpful for deadlock detection.

The applications of snapshots

3. **Loss of tokens.** Consider an algorithm that circulates tokens among processes, and processes may consume tokens. The property «There are at most k tokens» is stable, because tokens may be consumed, but not generated.

The applications of snapshots

3. **Loss of tokens.** Consider an algorithm that circulates tokens among processes, and processes may consume tokens. The property «There are at most k tokens» is stable, because tokens may be consumed, but not generated.
4. **Garbage collection.** In some programming environments a collection of objects is created, each of which may hold a *reference* to other objects. An object is called *reachable* if a path can be found from some designated object to this object by following references, and *garbage* otherwise.
   References may be added and deleted, but a reference to a garbage object is never added.
   Therefore, once an object has become garbage, it will remain garbage forever.

КОНЕЦ ЛЕКЦИИ 11.