Modular Approach for Developing Robust Protocols

Danny Dolev

The Hebrew University

a joint work with M. Ben-Or and E. Hoch

Ensemble

- A group communication system
- Modular and multi-layer structure
- Bob and his team used Nuprl to argue about the correctness of the modules in Ensemble
- I will describe a modular protocol overcoming transient and Byzantine faults -- proving its correctness using formal methods is a real **challenge**

Fast Self-Stabilizing Byzantine Tolerant Digital Clock Synchronization

- Clock synchronization
- Digital clock synchronization
- Self-stabilizing
- Byzantine tolerant
- Fast?

Model

- n nodes.
- Communication via message passing.
  - Private channels
  - No broadcast channels
- Network is fully connected.
- Synchronous (global beat system).
  - No round numbers
  - Round = between two consecutive beats
- Up to a third of the nodes may be Byzantine.
  - $n>3f$
  - No computational bounds and no cryptography
  - Adaptive and can “rush”
- Self-stabilizing.

Problem Definition

- The system is *clock_synched* at beat r with value $Clock(r)$ if, for each correct node u, it holds that $u.clock$ equals $Clock(r)$.

- The *k-Clock* problem is:
  Starting from any state, eventually (at some beat r) the system becomes *clock_synched* with value $Clock(r)$; and from this point on, at beat $r+i$ the system is *clock_synched* with value $Clock(r) + i \pmod{k}$.

Previous Work - Models

- Two avenues:
  - Deterministic / probabilistic
  - Timing model
- Timing model:
  - Global beat system (a.k.a. "synchronous")
  - Bounded-delay (a.k.a. "semi-synchronous")
- Bounded-delay has a slightly different problem definition.

Previous Work – Convergence Time

	synchronous	semi-synchronous
Probabilistic	●	●
Deterministic	●	●

Previous Work - Models

	synchronous	semi-synchronous
Probabilistic	$O(1)$	$O\left(n^{6(n-f)}\right)$
Deterministic	$O(f)$	$O(f)$

DW95, DDP03, HDD06, DH07

Overview of Solution -

- Assume the existence of a constant round, Byzantine tolerant, coin-flip module (Feldman-Micali, Katz-Koo)
- Create a “stream” of random bits.
- Construct a *2-Clock* from the random stream.
- Construct a *4-Clock* using two copies of *2-Clock*.
- Construct a *k-Clock* using one copy of *4-Clock* and the random stream.

What is a common-coin?

- (Not self-stabilizing)
- A distributed algorithm the has a binary *bit* output at each correct node.
- With some probability, $bit=1$ at all correct nodes, and with some probability, $bit=0$ at all correct nodes.
- With some probability *bit* might be different at different correct nodes.
- Example for poor probability: each node privately selects a random bit.

A "stream" of random bits

- Take any common-coin algorithm that:
  - is Byzantine tolerant
  - operates in a synchronous model
  - terminates within constant time
  - outputs a random bit, with **constant** probability
  - requires no special initialization
    - pre-agreed constants are allowed

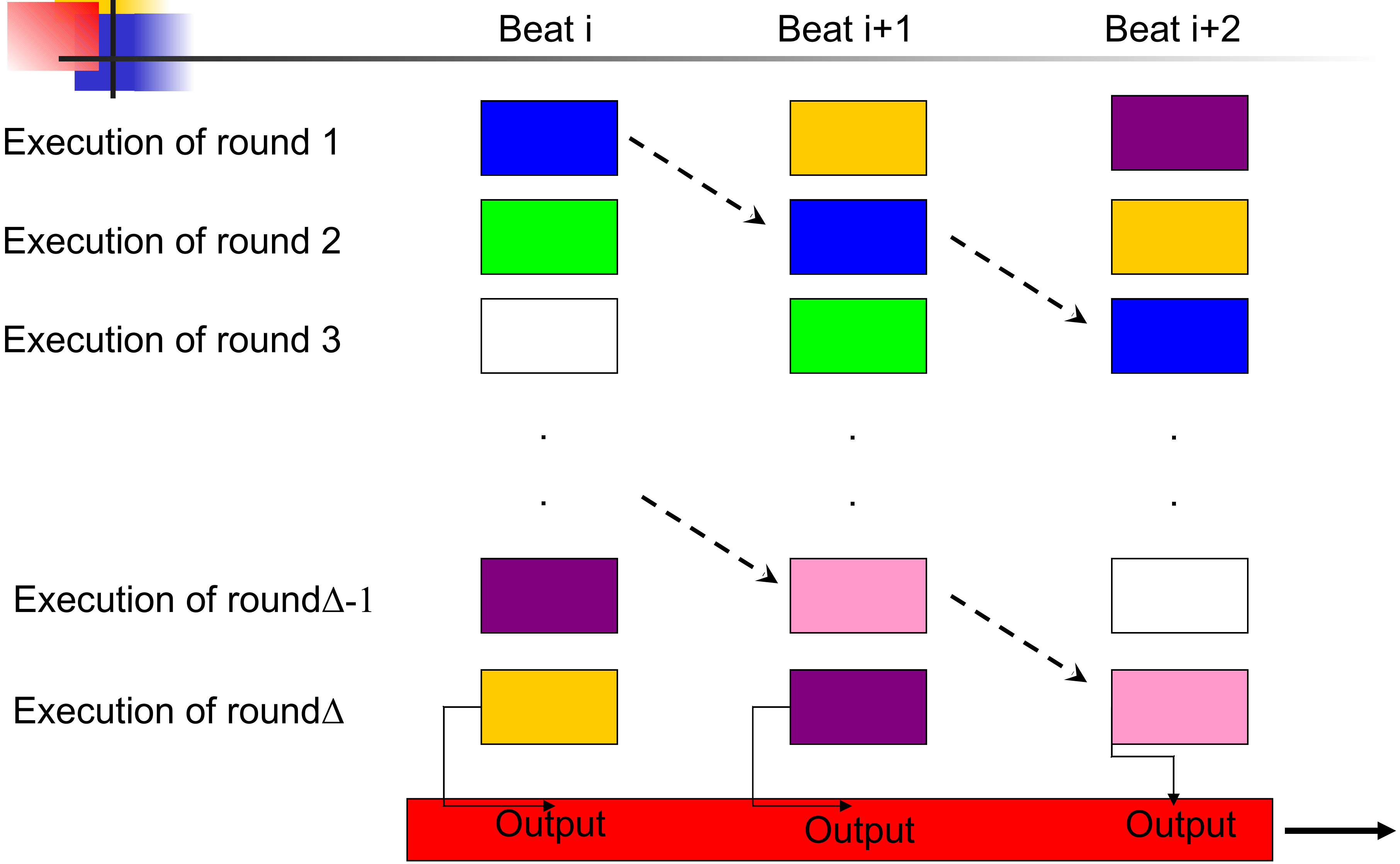
- Remark: need not be self-stabilizing.

A “stream” of random bits

- [FM89] provides a common-coin that terminates within O(1) rounds, and with constant probability produces a random bit, while supporting $n > 3f$.

- For the stream of random bits:
  - use pipelining to create a self-stabilizing Byzantine tolerant algorithm that produces a random bit every round.

The pipeline



Create a “stream” of random bits

Algorithm SS-BYZ-COIN-FLIP /* *executed at each node, each beat* */

/* $\mathcal{A}$ *is a probabilistic coin-flipping algorithm* */

/* *the* A_i*'s are* $\Delta_{\mathcal{A}}$ *instances of* $\mathcal{A}$ */

On beat (signal from global beat system):

1. **For** $i := 1$ to $\Delta_{\mathcal{A}}$
   execute the ith round of A_i;
2. **Output** the value of $A_{\Delta_{\mathcal{A}}}$;
3. **For** $i := 1$ to $\Delta_{\mathcal{A}} - 1$
   $A_{i+1} := A_i$;
4. **Initialize** A_1 to be a new instance of $\mathcal{A}$;

When you say "random", what do you mean?

- Common-coin algorithms usually have 3 stages: share, decide, recover:

>1 rounds
  - share + decide : at the end of these stages the "random" output is determined; however, no set of f nodes can retrieve it.

1 round
  - Recover : at the end of this stage, all nodes know the "random" output.

- Prior to the recovery stage, the Byzantine nodes do not know the output bit.
  During the recovery stage, the output bit is discovered.

How to reach binary consensus using a common coin?

How to reach binary consensus using a common coin?

- Goal: reach consensus on non-faulty nodes' input values.
- Method: each round, "do something" such that:
  - if all nodes agree on the consensus-value at the beginning of the round, they continue to do so;
  - if not, then with some probability all non-faulty nodes agree on the output value.
- Do this "forever".
  - eventually all non-faulty nodes will agree; and will stay so forever

How to reach binary consensus using a common coin?

- "Do Something":
  - Send input value to all nodes
  - If received n-f copies of the same value, take it as output
  - Otherwise, take the common coin as the output
  - (consider output of this round to be the input of the next round)
- "Proof": if some node received n-f copies of "1", then no one received n-f copies of "0". Thus, if the random bit is "1", all nodes have the same output.

Back to *2-Clock*…

Construct a *2-Clock* from the random stream

- Intuitive concept:
  - each round, send the value $(1 - u.clock)$ to all
  - if received less than $(n - f)$ copies of the same value, use the random bit as the new value
- Problem: adversary might be aware of the current random bit **before** it sends its "clock messages" of the current round. Thus, if the random bit is "1", it can send clock value "0"; preventing the non-faulty nodes to ever agree on the clock value.

Construct a *2-Clock* from the random stream

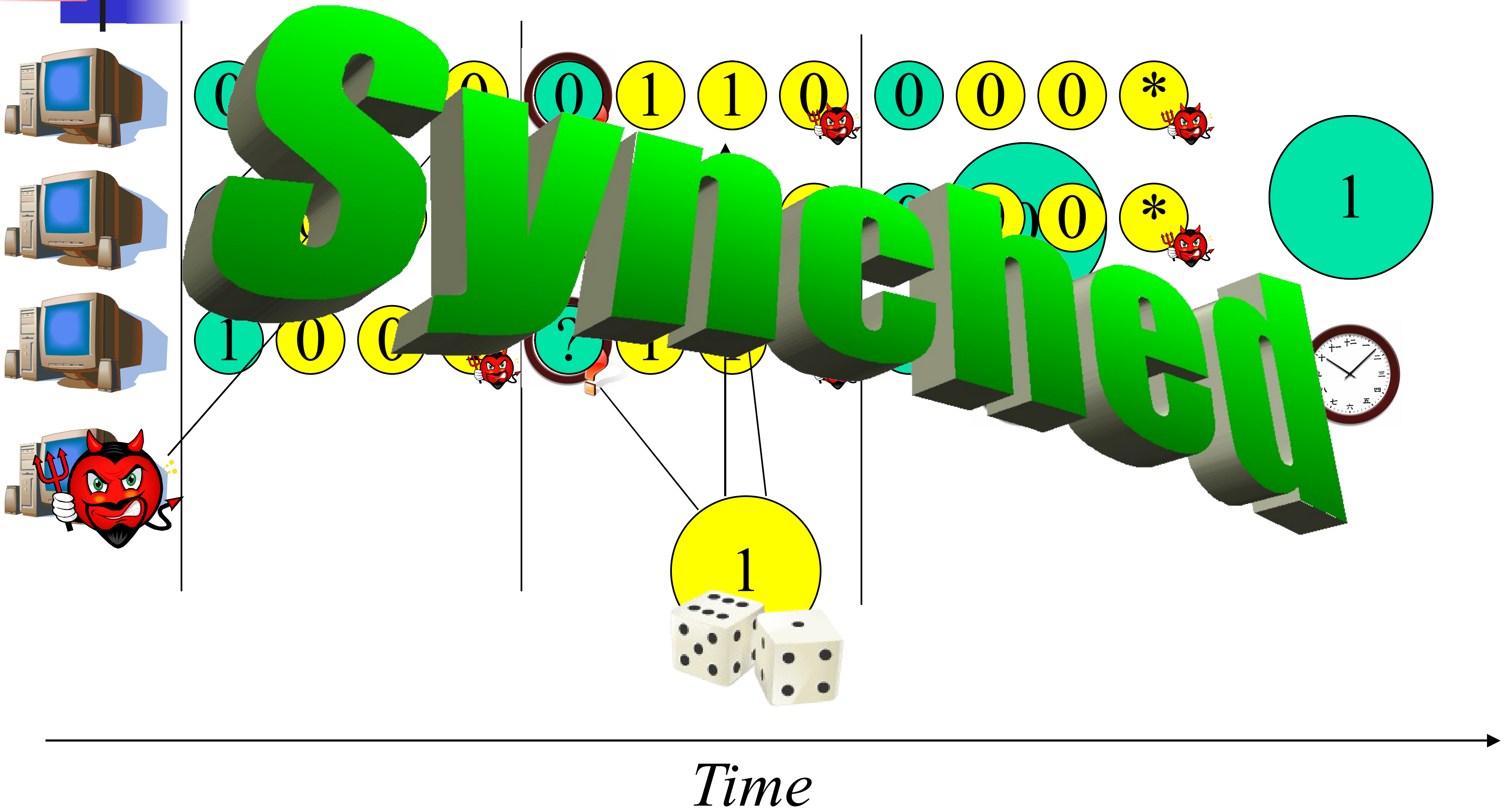
- Solution: use the random bit output regarding messages from the previous round only.
  - each round, send *1-clock* value (possibly "?") to all nodes
  - consider received messages with "?" as carrying the value of the next random bit
  - if received less than n-f copies of the same value, set the new clock value to "?"
- Disclaimer: messages sent the current round may depend of the random bit, but not messages sent in previous rounds!!!

Construct a *2-Clock* from the random stream

- End of round i: the set of node values is $\{v, ?\}$ for some specific v.
- Round $i+1$:
  - Each correct node sends "v" or "?".
  - Each correct node considers "?" as "*rand*".
    (With constant probability $rand := v$)
  - Each correct node checks if it has n-f copies of "v".
    - If *true*, set $u.clock :=$ "1-v";
    - Else, set $u.clock :=$ "?".

    (With constant probability, all correct nodes set $u.clock :=$ "1-v")

2-Clock example



Construct a *2-Clock* from the random stream

Algorithm SS-BYZ-2-CLOCK /* *executed at node u each beat* */

/* *$\mathcal{C}$ is self-stabilizing probabilistic coin-flipping algorithm* */

On beat (signal from global beat system):

1. **broadcast**[a] $u.clock$; /* $u.clock \in \{0, 1, \perp\}$ */
2. **execute** a single beat of $\mathcal{C}$, and set *rand* to be the output of $\mathcal{C}$;
3. **consider** each message with "$\perp$" as carrying the value *rand*; /* $rand \in \{0, 1\}$ */
4. **set** maj to be the value that appeared the most, and $\#maj$ the number of times it appeared; /* $maj \in \{0, 1\}$ */
5. **if** $\#maj \geq n - f$ then $u.clock := 1 - maj$;
6. **else** $u.clock := \perp$;

[a] In the context of this paper, "broadcast" means "send the message to all nodes". (We do not assume broadcast channels.)

Construct a *2-Clock* from the random stream

- Each round there is a constant probability that the system converges.
- Therefore, the convergence time is constant in expectation.
- Moreover, the probability that the system does not converge decreases exponentially.

Construct a *4-Clock* using two copies of *2-Clock*

- Basic idea: run 2 copies of *2-Clock*, one for the most significant bit and one for the least significant bit.
- Convergence time is still constant in expectation.

Construct a *4-Clock* using two copies of *2-Clock*

Algorithm ss-Byz-4-Clock /* *executed at node u each beat* */

/* $\mathcal{A}_1, \mathcal{A}_2$ *are instances of* ss-Byz-2-Clock */

On beat (signal from global beat system):

1. **execute** a single beat of $\mathcal{A}_1$;
2. **if** $u.clock(\mathcal{A}_1) = 0$ then
   execute a single beat of $\mathcal{A}_2$;
3. **set** $u.clock := 2 \cdot u.clock(\mathcal{A}_2) + u.clock(\mathcal{A}_1)$[a];

---

[a]To differentiate between the output *clock* value of ss-Byz-4-Clock and that of ss-Byz-2-Clock, consider $u.clock$ to be the output of ss-Byz-4-Clock, $u.clock(\mathcal{A}_1)$ is the output of $\mathcal{A}_1$ and $u.clock(\mathcal{A}_2)$ is the output of $\mathcal{A}_2$.

Construct a *k-Clock* using one copy of *4-Clock* and the random stream

- Could repeat the same "trick" as for the *4-Clock*; however, it would incur a logarithmic overhead in convergence time and message complexity.
- Instead, using a single instance of *4-Clock*, there are 4 phases of send-and-receive.
  - Use Turpin-Coan's 3-phase protocol to reach a multi-valued consensus from a binary consensus;
  - Use a 1-phase probabilistic protocol to reach a binary consensus;
  - In 4-phases there is a constant probability that the nodes agree on a clock value.
- The *4-clock* and *k-clock* values are not related.

Summary of Solution

- Assume the existence of a constant round coin-flip module (Feldman-Micali, Katz-Koo)
- Create a “stream” of random bits.
- Construct a *2-Clock* from the random stream.
- Construct a *4-Clock* using two copies of *2-Clock*.
- Construct a *k-Clock* using one copy of *4-Clock* and the random stream.

Overview of the other modules

- Create a constant round coin-flip module.
- Oblivious Leader Election.
- Moderated VSS.
- Grade-cast (a weak form of agreement).
- Verifiable Secret Sharing (assuming broadcast).

Bob !!! – back to work

With occasional resting

Thank you!