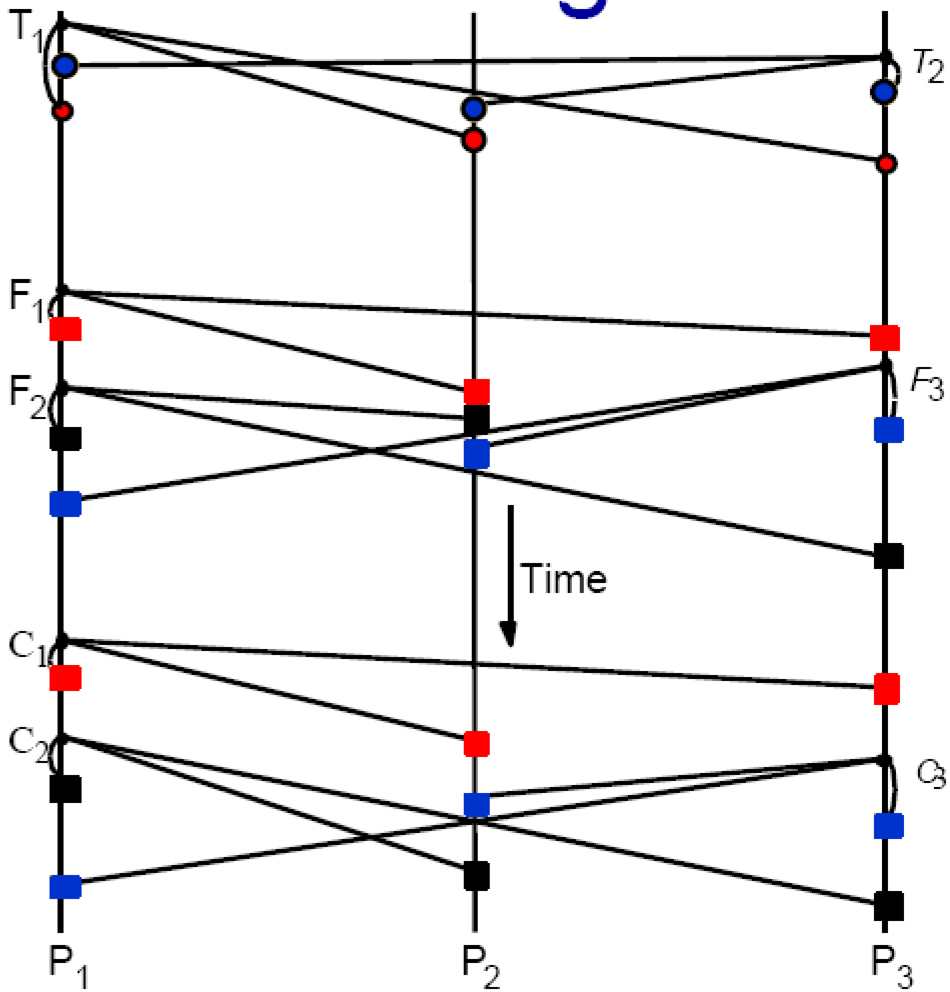
Ordered Multicast

- What if ordering of messages is critical to system?
  - basic algorithm does not make guarantees about order
- Ordered Multicast
  - FIFO Ordering
    - If correct process issues multicast(g,m) and then multicast(g,n), then every correct process that delivers n will deliver m before n
  - Causal Ordering
    - If multicast(g,m) $\rightarrow$ multicast(g,n), where $\rightarrow$ is restricted to group g messages, then every correct process that delivers n will deliver m before n
  - Total Ordering
    - If correct process delivers m before n, then every correct process that delivers n will deliver m before n

Total, FIFO, Causal Ordering of Multicast Messages

- consistent ordering of totally ordered messages T_1 and T_2
- FIFO-related messages F_1 and F_2
- causally related messages C_1 and C_3
  - otherwise arbitrary delivery ordering of messages.



Ordering Relationships

- FIFO Ordering is a partial order
  – messages of different processes
- Causal Ordering is a partial order
  – concurrent multicasts
- Causal Ordering implies FIFO Ordering
  – multicasts of same process always causally related
- Total Ordering is independent of FIFO and Causal Orderings
  – can define hybrid FIFO-Total and Causal-Total Orderings
- Ordered multicast can be unreliable
  – p delivers m and then n, q delivers m but not n
  – hybrids of ordered and reliable protocols
  – atomic multicast
    - reliable and total

Bulletin Board Example

- What kind of multicast delivery guarantees might be useful?
  - Reliable
    - every user sees every message eventually
  - FIFO
    - same user's messages ordered correctly
  - Causal
    - threads have correct ordering
  - Total
    - consistent numbering of messages
  - What do real-life bulletin boards guarantee?
    - nada, nothing, zilch
    - post in order received - Why?

Bulletin Board Program

	Bulletin Board	*os.interesting*
Item	From	Subject
23	A. Hops	Mach
24	B. Moss	Microkernel
25	C. Chops	Re: Mach
26	A. Hops	RPC
27	D. Snobs	Re: RPC

Implementing FIFO Ordering

- How can we achieve FIFO-ordered multicast?
  - sequence numbers
  - Algorithm B for reliable multicasting obeys FIFO ordering
  - Can we enforce FIFO on top of any basic multicast?
- Similar to algorithm B except for using basic B-multicast
  - use sequence number piggy backed on messages
  - hold back any message that is future sequence
  - overlapping groups?
  - reliability?

Implementing Total Ordering

- How can we achieve Total-ordered multicast?
  - totally ordered sequence numbers
  - delivery algorithm same as FIFO ordering
  - group-specific sequence numbers instead of process-specific
- Possible Implementations
  - use sequencer process to assign unique number that is piggy backed on messages
  - distributed agreement on sequence numbering
  - overlapping groups?

Total Ordering with Sequencer

1. Algorithm for group member *p*

On initialization: $r_g := 0$;

To TO-multicast message m to group g

   B-multicast$(g \cup \{sequencer(g)\}$, <*m*, *i*>);

On B-deliver(<*m*, *i*>) *with g = group*(*m*)

   Place <*m*, *i*> in hold-back queue;

On B-deliver(<“order”, *i*, *S*>) *with g = group*(*m*)

   wait until <*m*, *i*> in hold-back queue and $S = r_g$;

   TO-deliver m;   // (after deleting it from the hold-back queue)

   $r_g = S + 1$;

Total Ordering with Sequencer

2. Algorithm for sequencer of *g*

On initialization: $s_g := 0;$

On B-deliver(<*m*, *i*>) *with g = group*(*m*)
    B-multicast(*g*, <"order", *i*, s_g>);
    $s_g := s_g + 1;$

Sequencer Issues?

Causal Ordering using Vector Timestamps

Algorithm for group member p_i $(i = 1, 2..., N)$

On initialization

$V_i^g[j] := 0$ $(j = 1, 2..., N)$;

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k]$ $(k \ne j)$;

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

Consensus and related problems

- Consensus
- Byzantine generals
- Interactive consistency

Consensus

- How do a group of processes come to a decision?
- Suppose a number of generals want to attack a target. They know that they will only succeed if they all attack. If anybody backs out then it is going to be a defeat.
- The example becomes more complicated if one of the generals becomes a traitor and starts to try and confuse the other generals. By saying yes I'm going to attack to one and no I'm not to another.
- How do we reach consensus when there are Byzantine failures? It depends on if the communication is synchronous or asynchronous

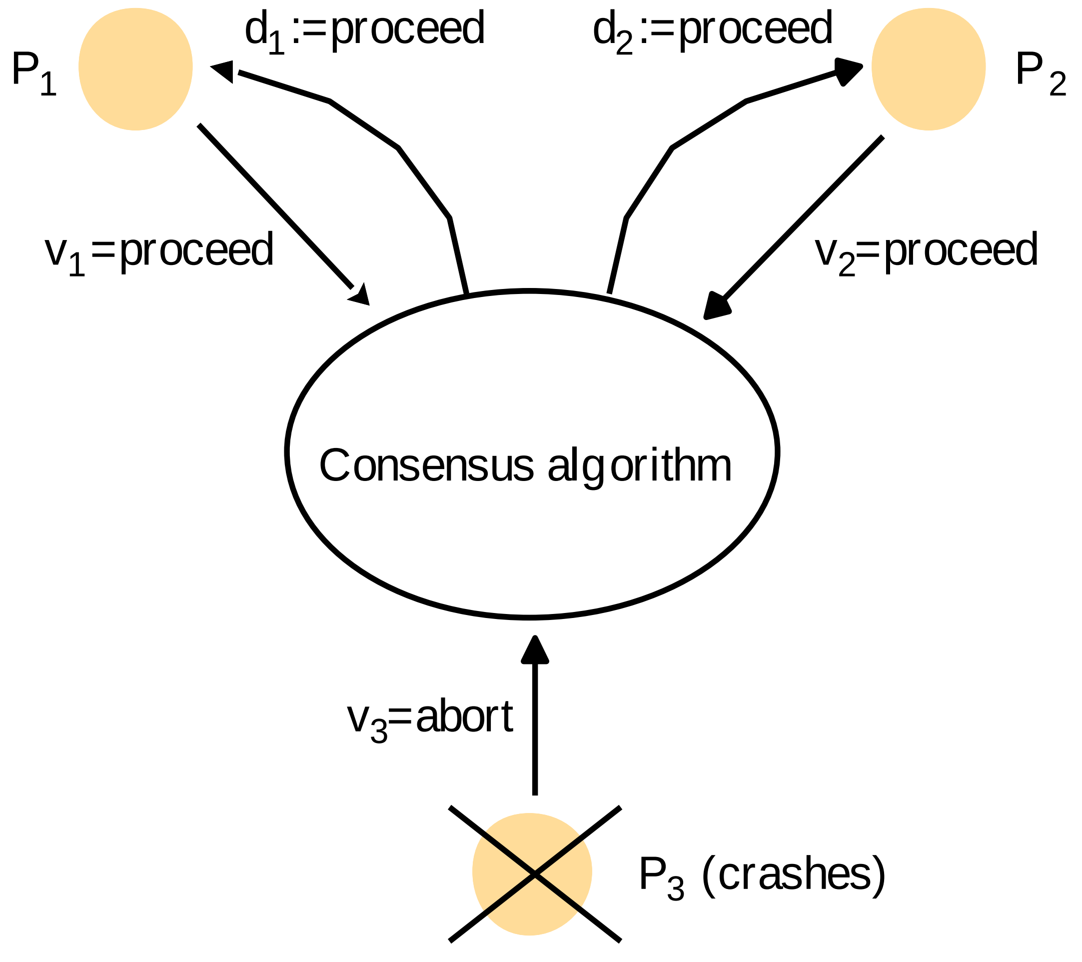
System model

- We have N processes $P=(p_1, p_2, \ldots, p_N)$
- Communication is reliable
- Processes may fail (arbitrary and crash)
- Assume that signing does not happen (digital signing makes it impossible for a faulty process to make a false claim about the values that a correct process has sent to it)

Problem definitions

- Each process p_i starts in the *undecided* state
- And then *proposes* a single value v_i from a set D ($i=1,2,\ldots,N$)
- The processes communicate with each other through exchanging values
- Each process then sets the value of a *decision variable* d_i
- It enters the *decided* state and may no longer change d_i

Consensus for three processes



Requirements

Agreement: correct processes can propose different values but eventually the decision value of all correct processes is the same.
Integrity: correct processes ALL proposed the SAME value, then any correct process in the decided state had chosen that value.

s

- Agreement: The decision value of all correct processes is the same

- Integrity: If the correct processes all proposed the same value, then any correct process in the *decided* state had chosen that value

Differences between Agreement and Integrity?

If processes cannot fail

The consensus problem is easy to solve

- Each process reliably multicast its proposed value to the *members* of the group
- Each process waits until it has collected all *N* values (including its own)
- It then evaluates the function majority(v_1, v_2, …, v_N), which returns the value that occurs most often among its arguments, or the special value not belong to *D*

All the three requirements are satisfied

If a process can crash or fail in arbitrary ways

- If processes can crash then it is not clear whether a run of the consensus algorithm can terminate (asynchronous)

- If processes can fail in arbitrary ways, then faulty processes can in principle communicate random values to the others

- In this case, correct processes must compare what they have received with what other processes claim to have received

Byzantine generals problem

- 3 or more generals are to agree to attack or to retreat
- One, the commander, issues the order
- The others are to decide to attack or retreat
- But one or more of the generals may be “treacherous” (faulty)
- If the commander is treacherous, he proposes attacking to one general and retreating to another
- If a normal general is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat

The three requirements

- Termination: Eventually each correct process sets its decision variable
- Agreement: The decision value of all correct processes is the same.
- Integrity: If the commander is correct, then all correct processes decide on the commander’s value

Difference from the general consensus?

- The difference here is that in the byzantine general problem a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value.

Interactive consistency

- Another variant of consensus, in which every process proposes a single value
- The goal is for the correct processes to agree on a *vector* of values (decision vector), one for each process
- For example, the goal could be for each of a set of processes to obtain the same information about their respective states

3 requirements

- Termination: Eventually each correct process sets its decision variable
- Agreement: The decision vector of all correct processes is the same.
- Integrity: If p_i is correct, then all correct processes decide on v_i as the *i*th component of their vector

Consensus in a synchronous system

- Use a basic multicast protocol
- Assume that up to *f* of the *N* processes exhibit crash failures (not arbitrary failures)
- To reach consensus, each correct process collects proposed values from the other processes
- The algorithm proceeds in *f*+1 rounds, in each of which the correct processes B-multicast the values between themselves
- The algorithm guarantees that at the end of the rounds all the correct processes that have survived are in a position to agree

Algorithm

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \leq r \leq f + 1)$

B-multicast(g, $Values_i^r - Values_i^{r-1}$); // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round *r*)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

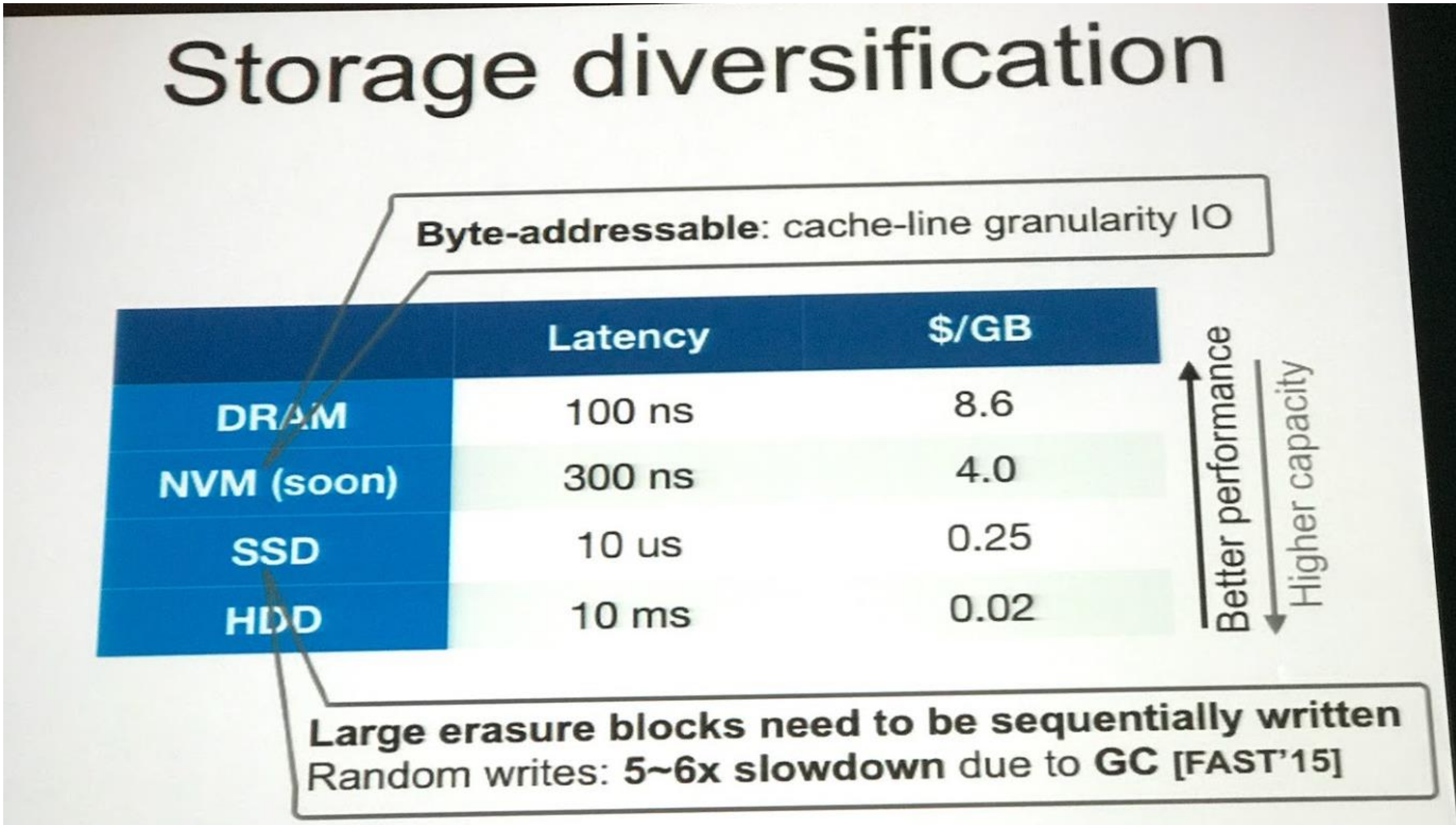
After $(f + 1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

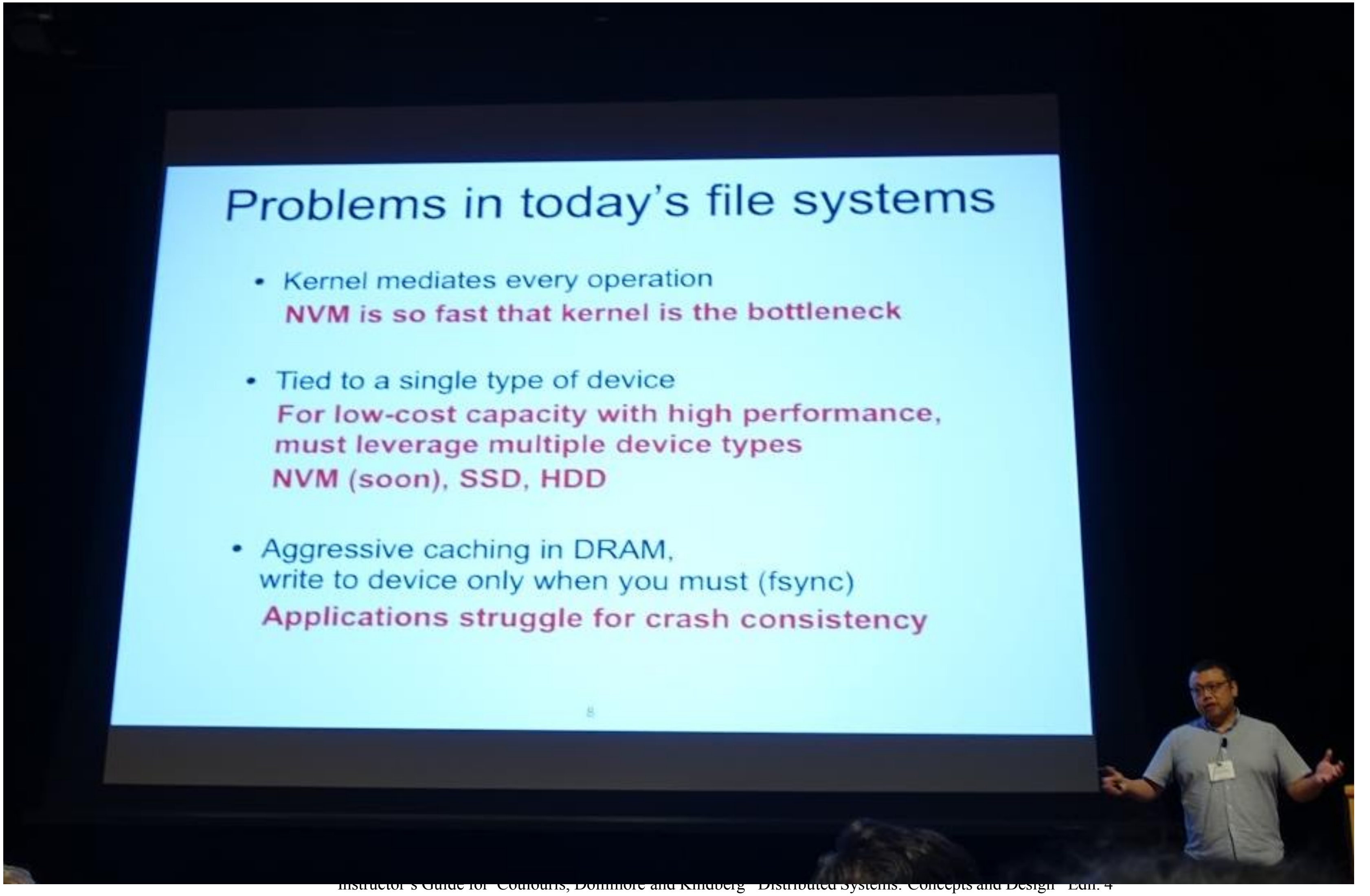
The three properties

- Termination is obvious because the system is synchronous
- Each process arrives at the same set of values at the end of the final round
- Thus, agreement and integrity will follow because the processes apply the *minimum* function to this set

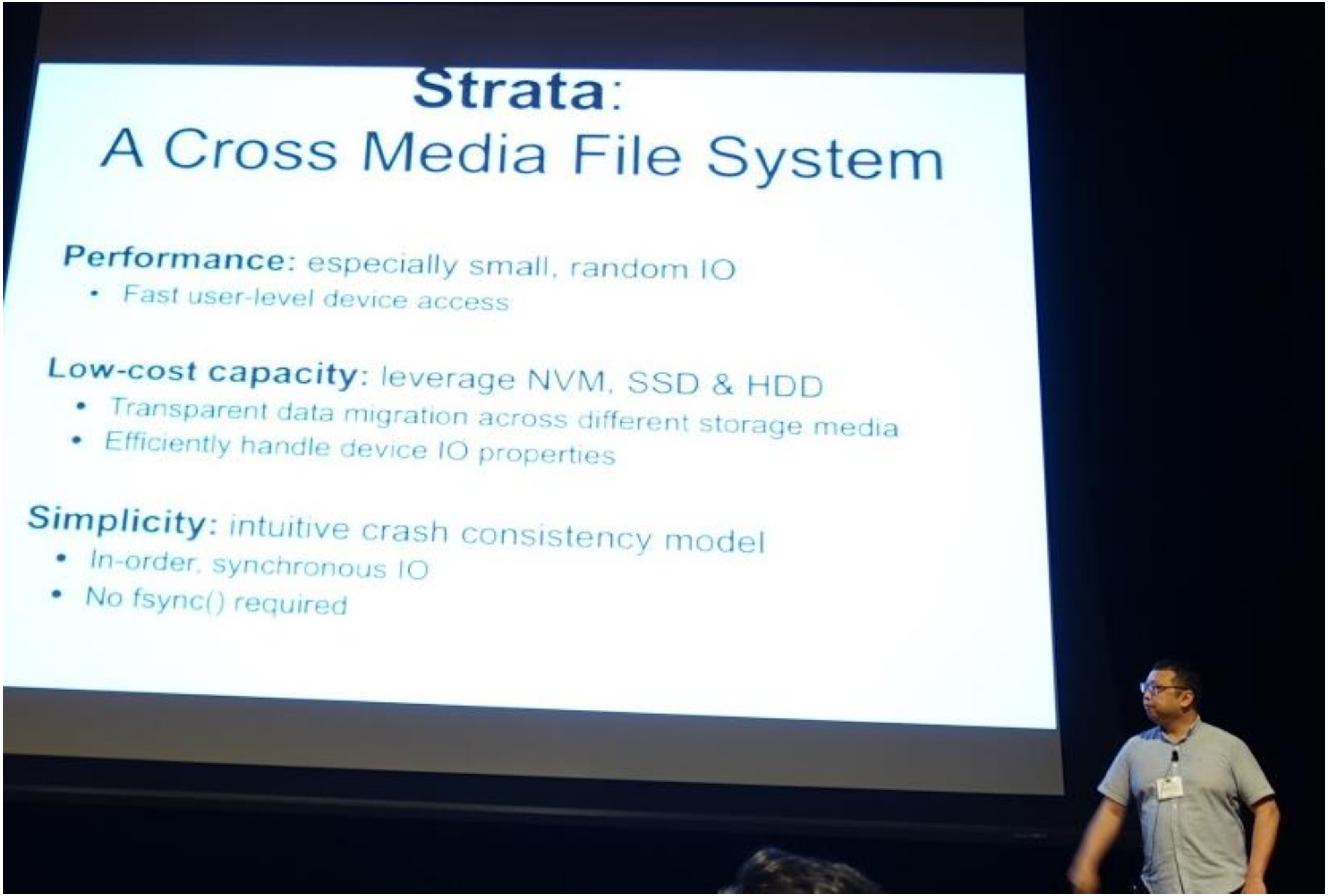
From NVMW 2018



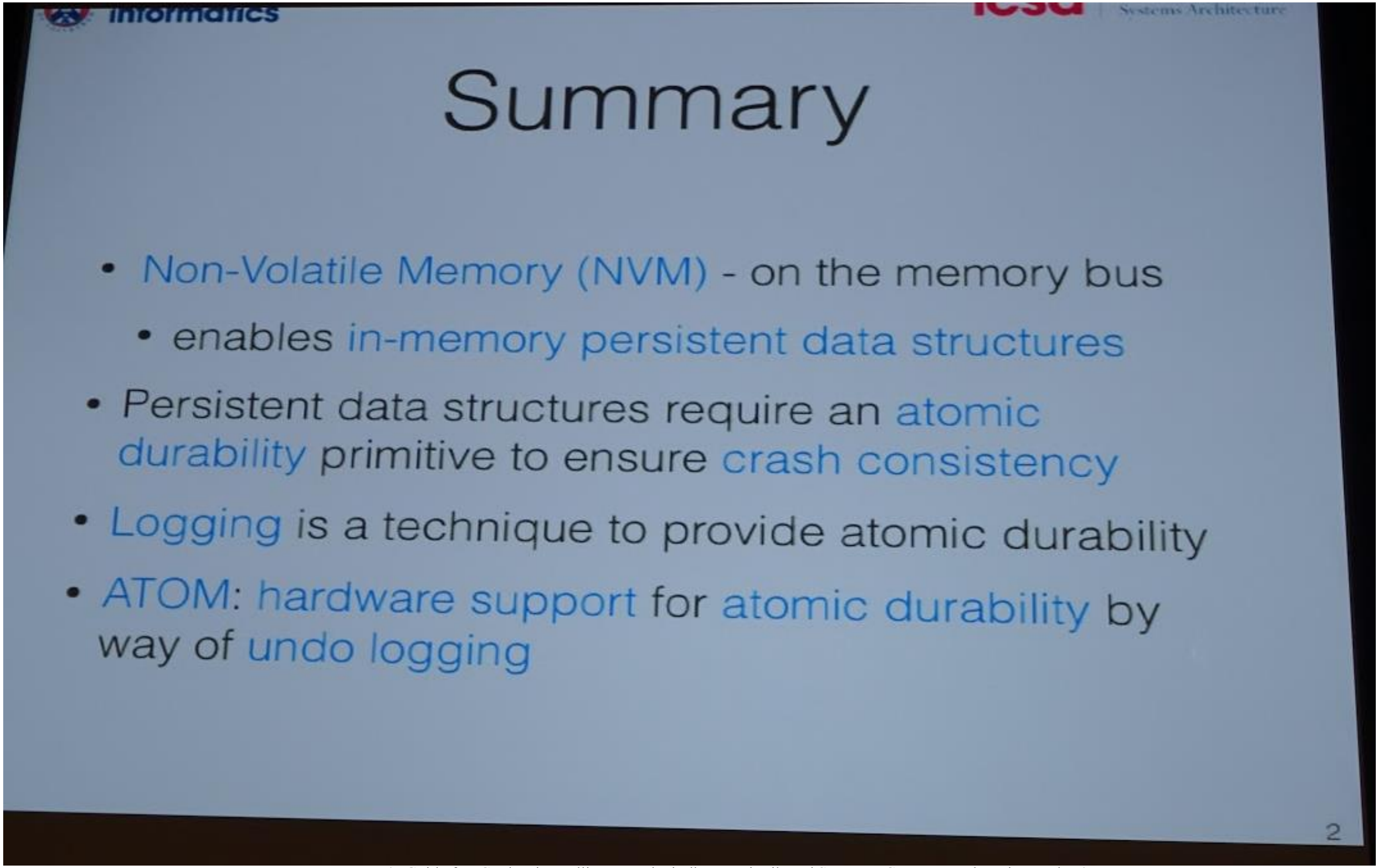
From NVMW 2018



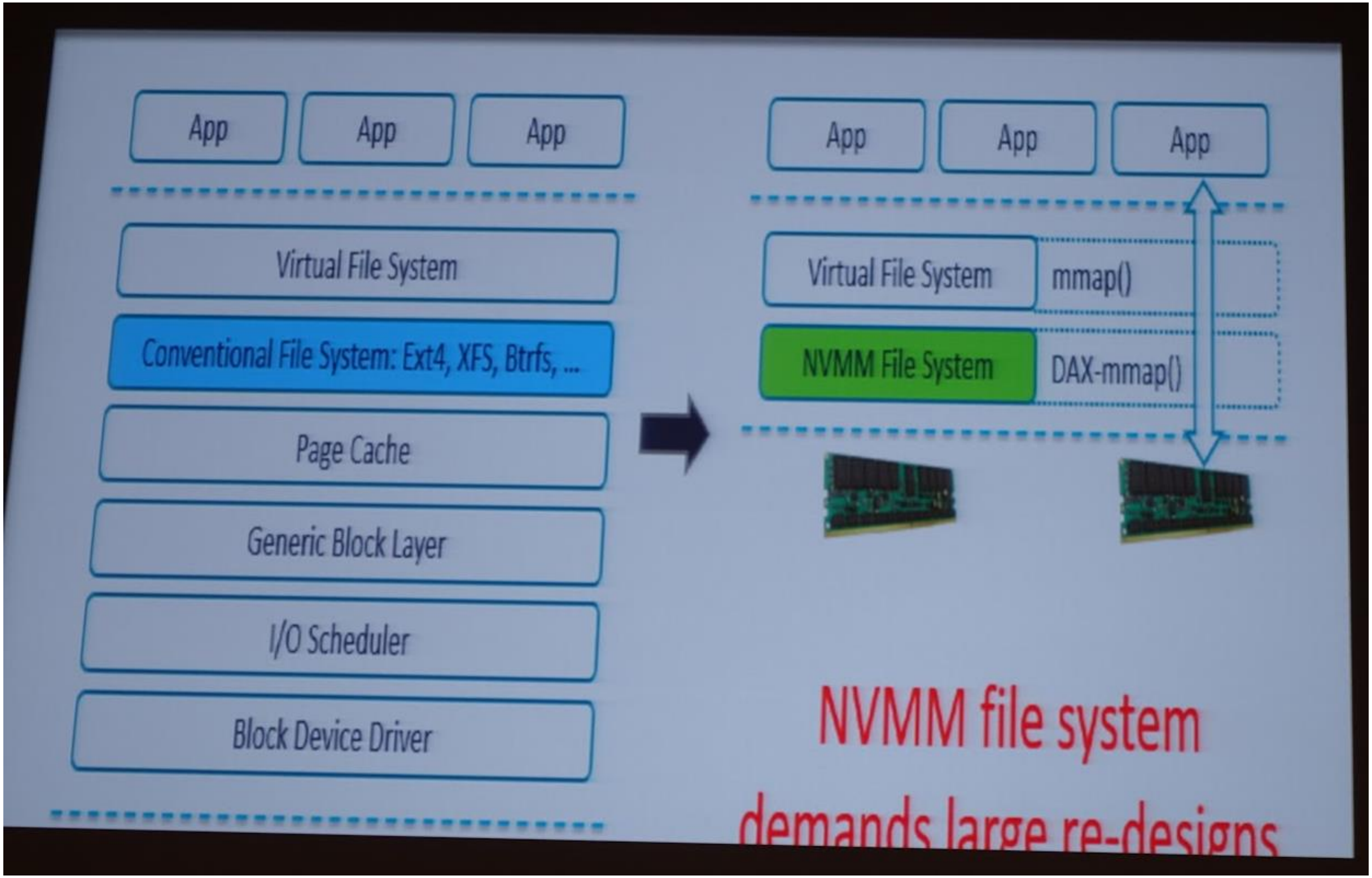
From NVMW 2018



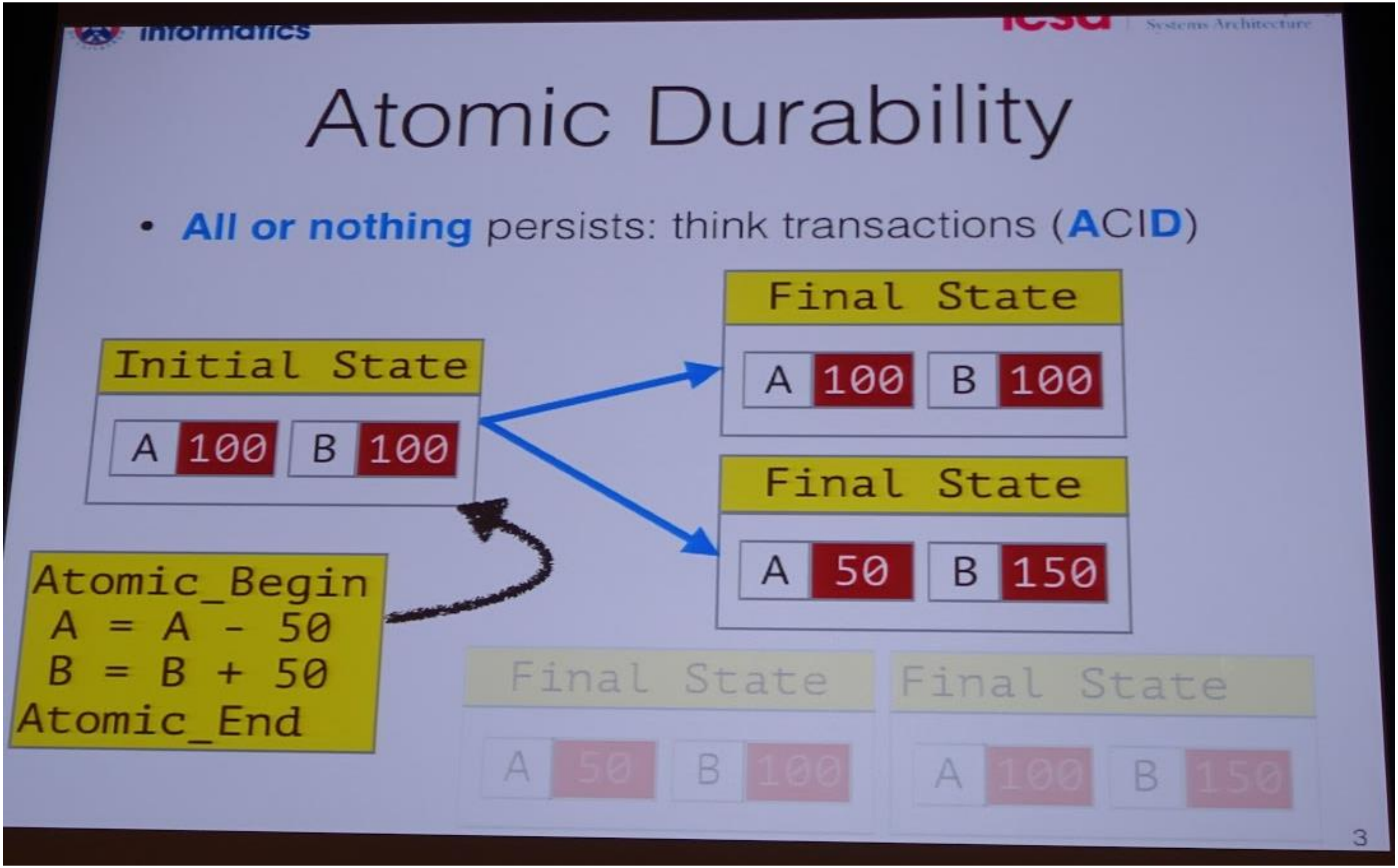
From NVMW 2018



From NVMW 2018



From NVMW 2018



From NVMW 2018