A Programming Model for Failure-Prone, Collaborative Robots

Nels Eric Beckman
Jonathan Aldrich

School of Computer Science
Carnegie Mellon University

SDIR 2007
April 14th, 2007

Failure Blocks: Increasing Application Liveness

- In the Claytronics domain, failure will be commonplace.
- Certain Applications:
  - Failure of one catom causes others to be useless.
- Our model:
  - An extension to remote procedure calls.
  - Helps developers preserve liveness.
  - Developers
    - Signify where liveness is a concern.
    - Specify liveness preserving actions.
  - When failure is automatically detected, those actions are taken.

Outline

- In our domain, the rate of failure will be high
- 'Hole Motion' (An example failure scenario)
- Existing RPC systems do not help us to preserve liveness
- Our model has two key pieces
  - The failure block
  - The compensating action

Rate of Failure in Catoms will be High

- Due to the large numbers involved:
  - Per-unit cost must be low, which implies
    - A lack of hardware error detection features.
    - Rate of mechanical imperfections will be high.
  - Probability of some catom failing becomes high.

- Interaction with the physical world:
  - Dust particles?
  - Other unintended interactions?

Outline

- In our domain, the rate of failure will be high
- ‘Hole Motion’ (An example failure scenario)
- Existing RPC systems do not help us to preserve liveness
- Our model has two key pieces
  - The failure block
  - The compensating action

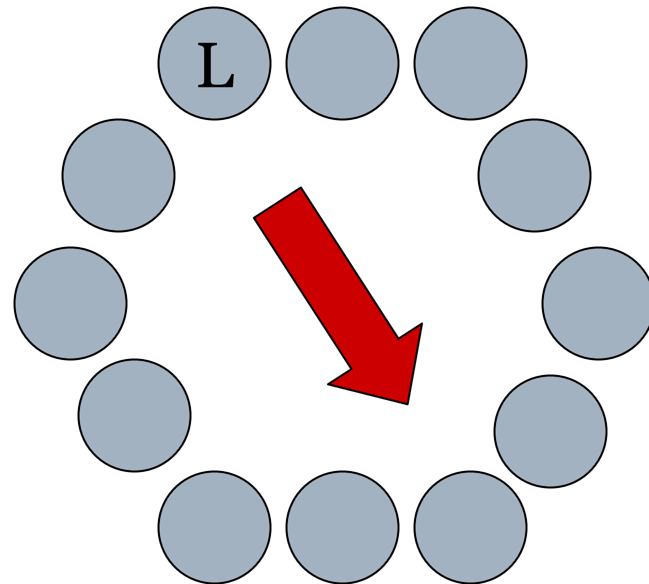
The Hole Motion* Algorithm

- A Motion-Planning Technique
- The Idea:
  - Randomly send holes through the mass of catoms.
  - Holes ‘stick’ to areas that should shrink.
  - They are more likely to be created from areas that should grow.

*De Rosa, Goldstein, Lee, Campbell, Pillai. Scalable Shape Sculpting Via Hole Motion: Motion Planning in Lattice-Constrained Modular Robots. *IEEE ICRA 2006*. May 2006.

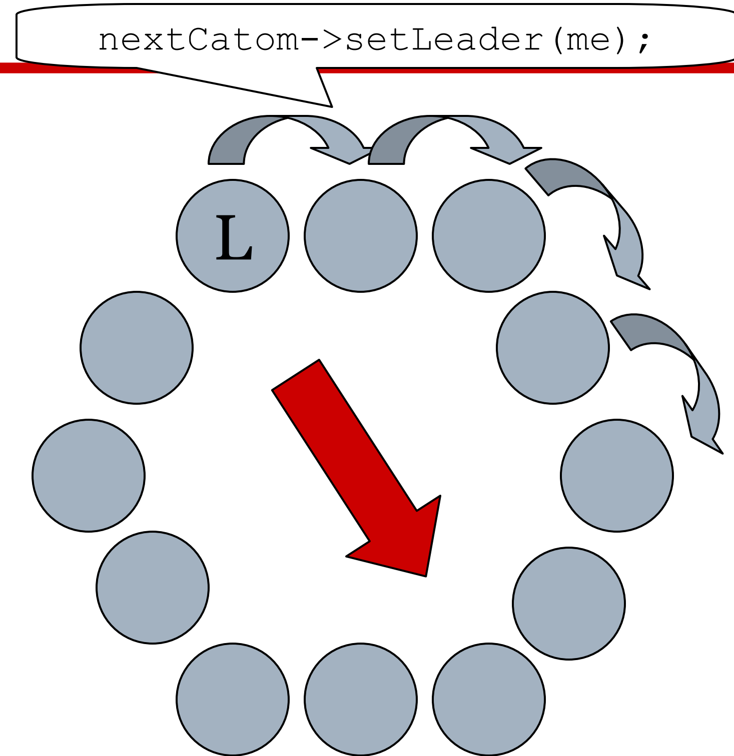
In Detail…

- At each 'hole time-step,' catoms around the hole have a leader.
- They only accept commands from this leader.
- This protects the hole's integrity.



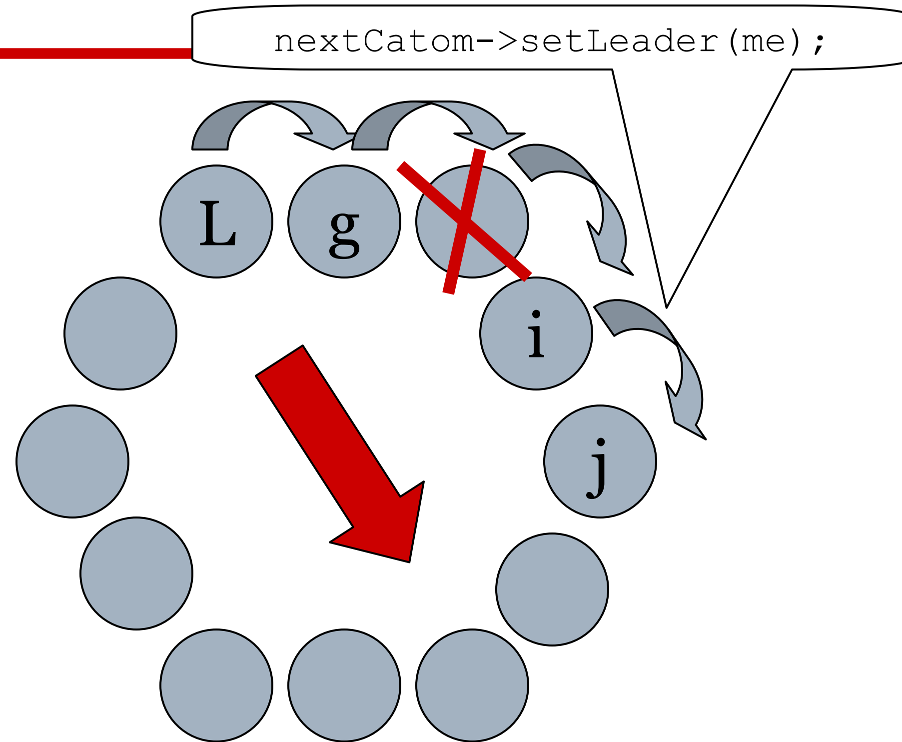
In Detail…

- In order to become the ‘leader,’ this catom calls ‘setLeader’ on its neighbors.
- The same method is called recursively on other would-be group members.



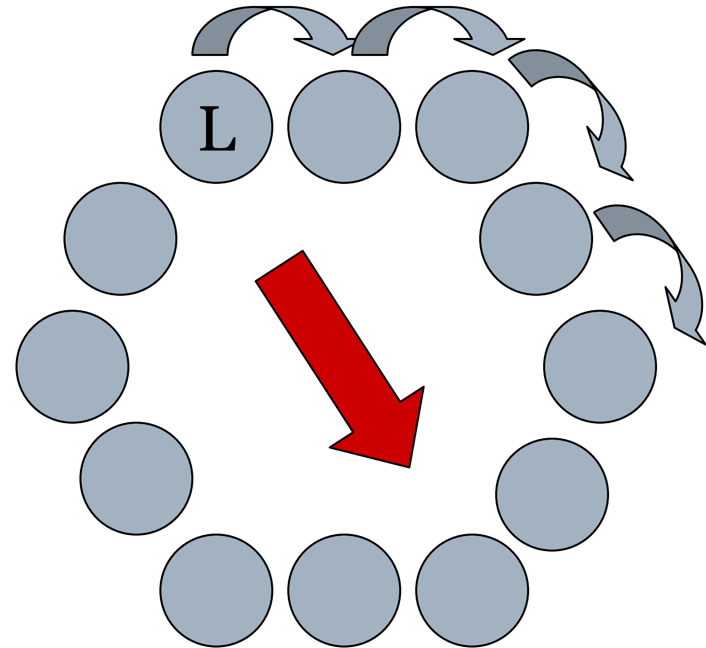
In Detail…

- Catom on the stack fails:
  - Catoms i and j may have already set L as their leader!
  - But the only communication path to L is gone.



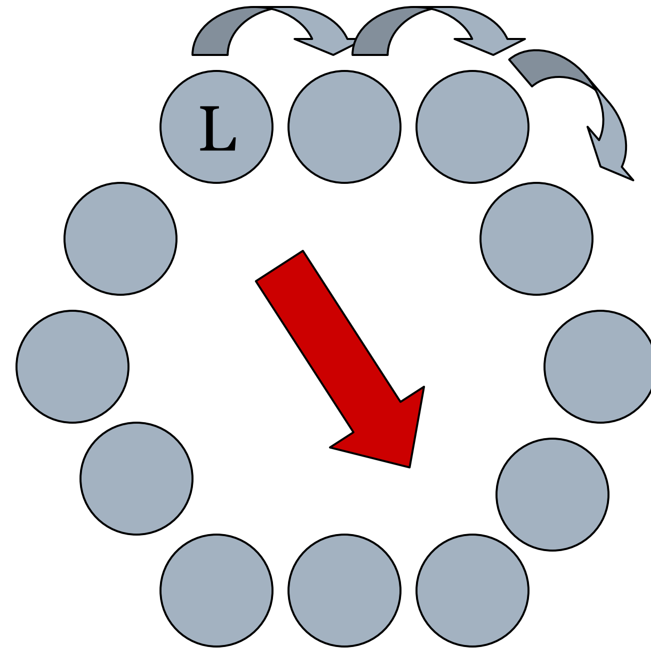
In Detail…

- Instead, suppose operation returned normally…



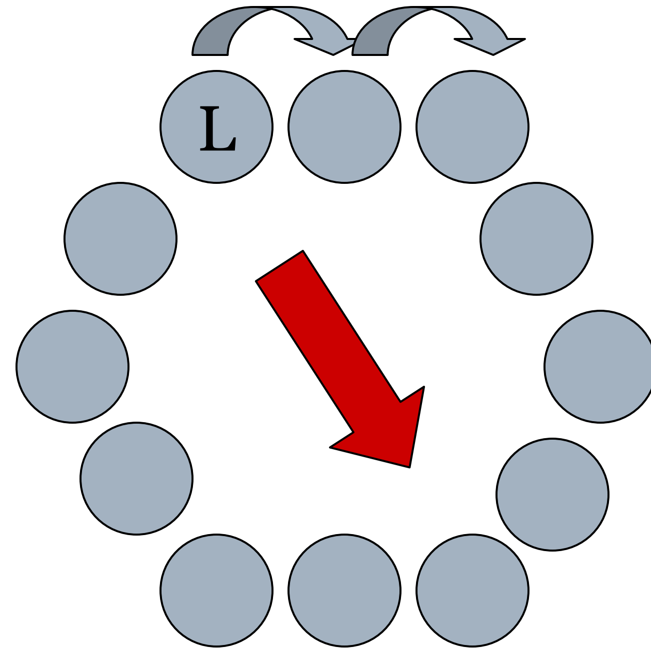
In Detail…

- Instead, suppose operation returned normally…



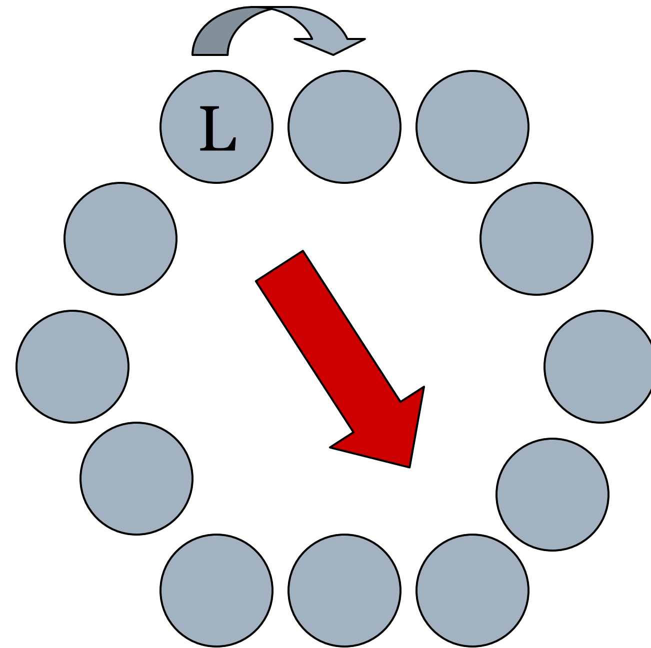
In Detail…

- Instead, suppose operation returned normally…



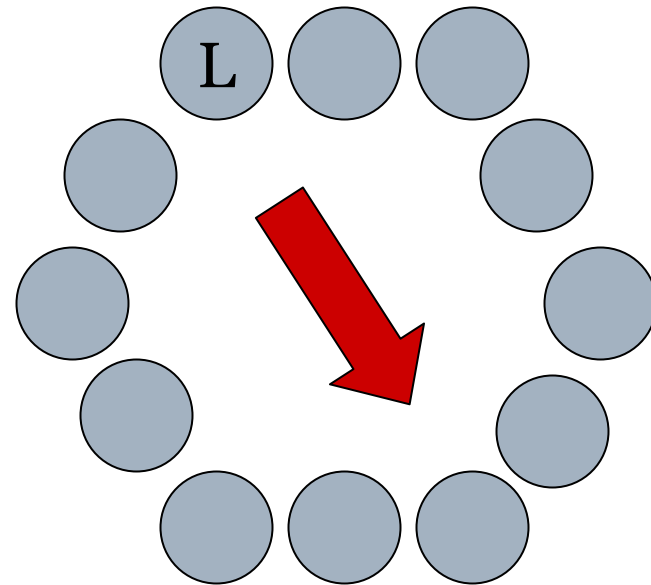
In Detail…

- Instead, suppose operation returned normally…



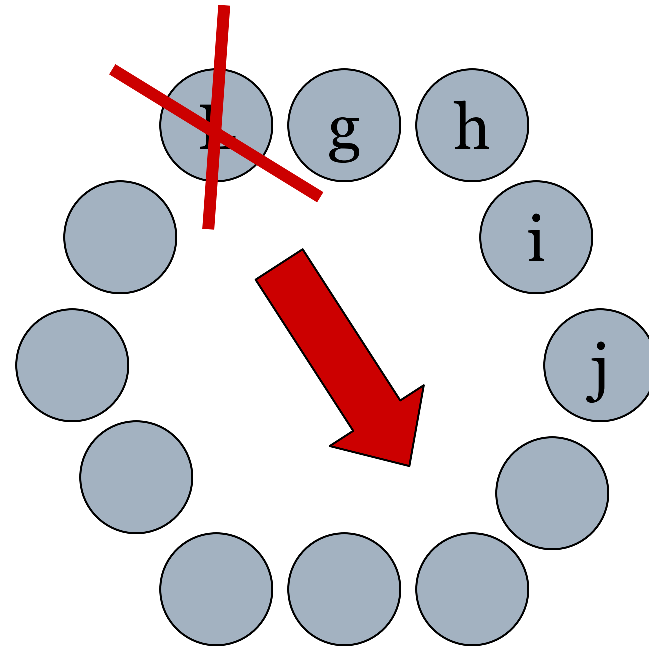
In Detail…

- Instead, suppose operation returned normally…



In Detail…

- Now L fails:
  - Catoms g-j (and all the rest) expect commands from L!
  - For all practical purposes, 12 catoms have failed.



Outline

- In our domain, the rate of failure will be high
- ‘Hole Motion’ (An example failure scenario)
- Existing RPC systems do not help us to preserve liveness
- Our model has two key pieces
  - The failure block
  - The compensating action

Existing RPC Systems, Not a Perfect Fit

- Weak Failure Detection
  - Usually a timeout mechanism.
  - Our model uses active failure detection.

- No Callee-Side Failure Handling
  - Caller can catch timeout exception; not callee.
  - But the callee could be left in an invalid state.
  - Our model provides callee with compensating actions.

Existing RPC Systems, Not a Perfect Fit

- Only detect failure on the stack of RPC calls.
  - Our model designates catoms as being a part of the group for a lexical ‘amount of time.’
  - They are still a part of this group when the thread moves to a different location.
  - Failures on the stack and off are dealt with in the same manner.

Outline

- In our domain, the rate of failure will be high
- 'Hole Motion' (An example failure scenario)
- Existing RPC systems do not help us to preserve liveness
- Our model has two key pieces
  - The failure block
  - The compensating action

The Model: Two Key Pieces

- `fail_block`, which specifies
  - The logical 'time period' during which liveness concerns exist
  - The members of the group (implicitly)
  - Where control should return in the event of a failure
- `push_comp`, which allows
  - The specification of code to be executed in the event of catom failure

The `fail_block` Primitive

- `fail_block` *b*
- Evaluates the code in block *b*.
- In the event of a *detected* failure
  - The entire block throws an exception.
  - Execution continues from the catom where the failure block is evaluated.

The `fail_block` Primitive

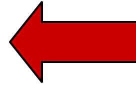
- At runtime, the entire operation is given a unique 'operation ID.'
  - When a RPC is called from within block
    - Callee becomes 'part' of the operation.
    - Callee and caller add one another as collaborators.
    - They 'ping' each other regularly to detect failure.
    - Applies recursively.
  - In the event a failure is detected, they share the information about the demise of that operation.

The `fail_block` Primitive

- If *b* is successfully executed
  - An ‘end’ message is sent out.
  - Collaborators stop detecting failure for that OID.

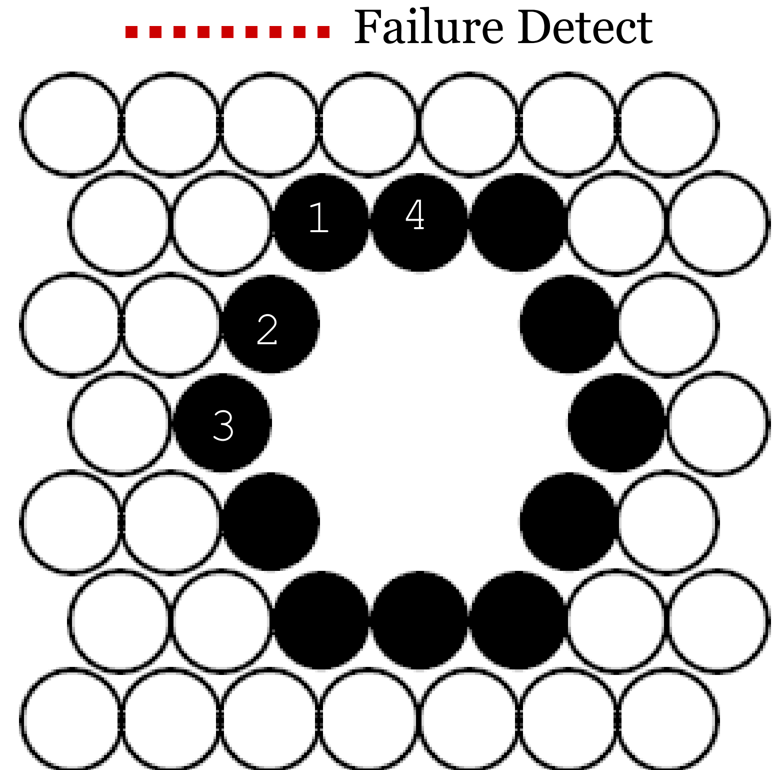
‘Demo’

```
fail_block {
  // catom 1
  lnode->setBoss(this);
  rnode->setBoss(this);
}
...
setBoss(Catom h)
  myLeader = h;
}
```



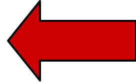
Group Members: {}

Op ID:



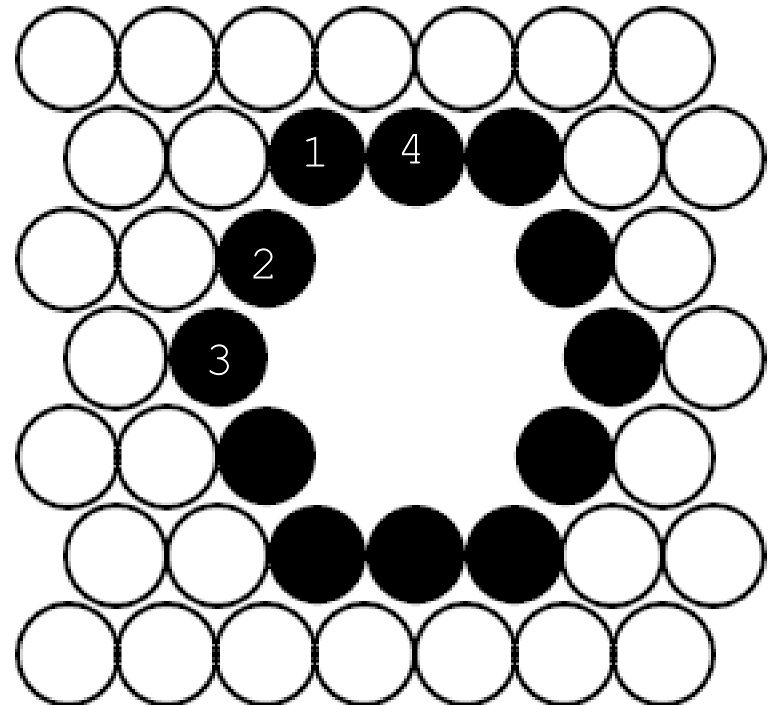
‘Demo’

```
fail_block {
  // catom 1
  lnode->setBoss(this);
  rnode->setBoss(this);
}
...
setBoss(catom h) {
  myLeader = h;
}
```



Group Members: {1}

Op ID: 23423123

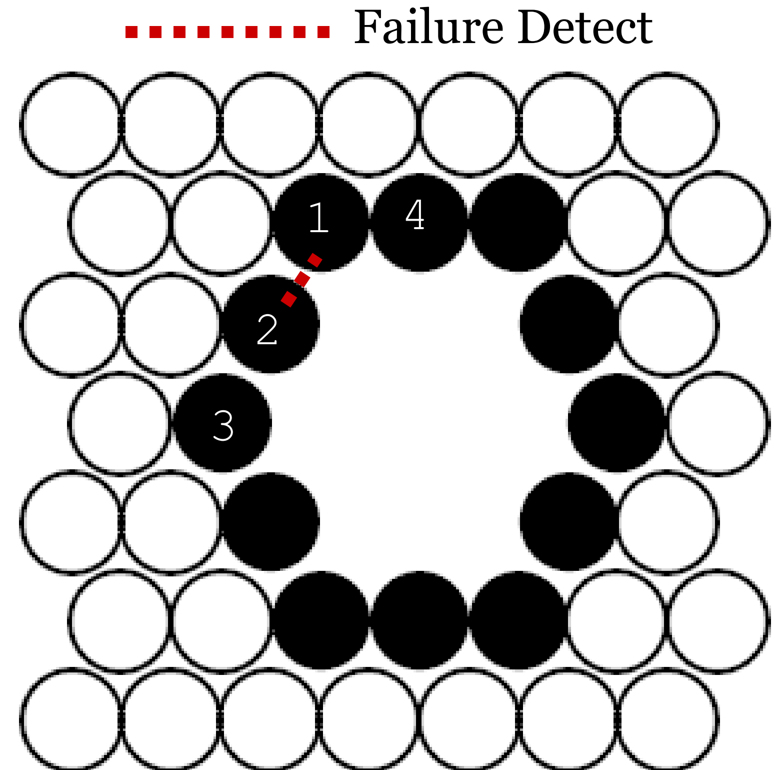


‘Demo’

```
fail_block {
  // catom 1
  lnode->setBoss(this);
  rnode->setBoss(this);
}
...
setBoss(catom h) {
  myLeader = h;
}
```

Group Members: {1,2}

Op ID: 23423123

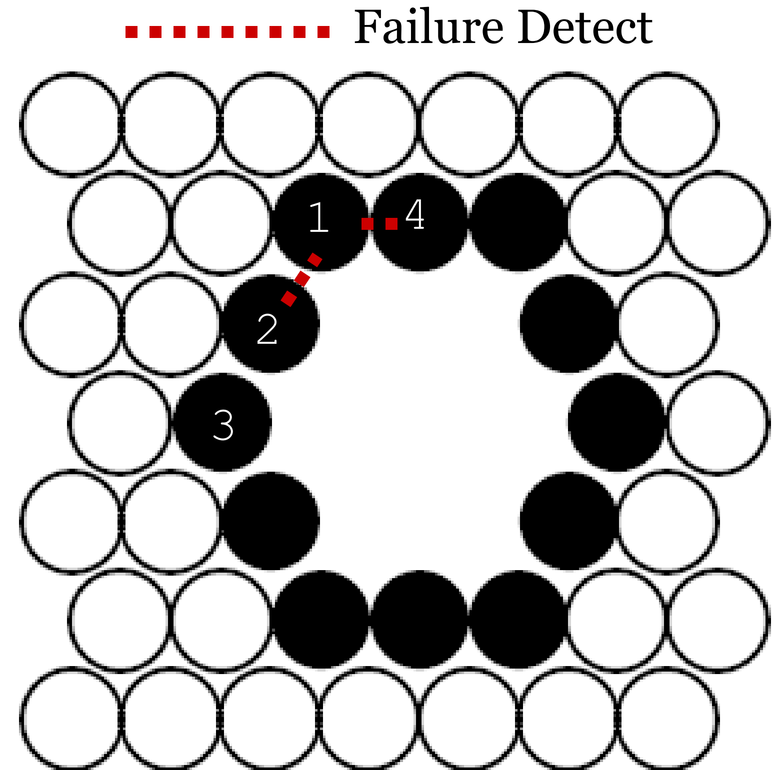


‘Demo’

```
fail_block {
  // catom 1
  lnode->setBoss(this);
  rnode->setBoss(this);
}
...
setBoss(catom h) {
  myLeader = h;
}
```

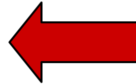
Group Members: {1,2,4}

Op ID: 23423123



‘Demo’

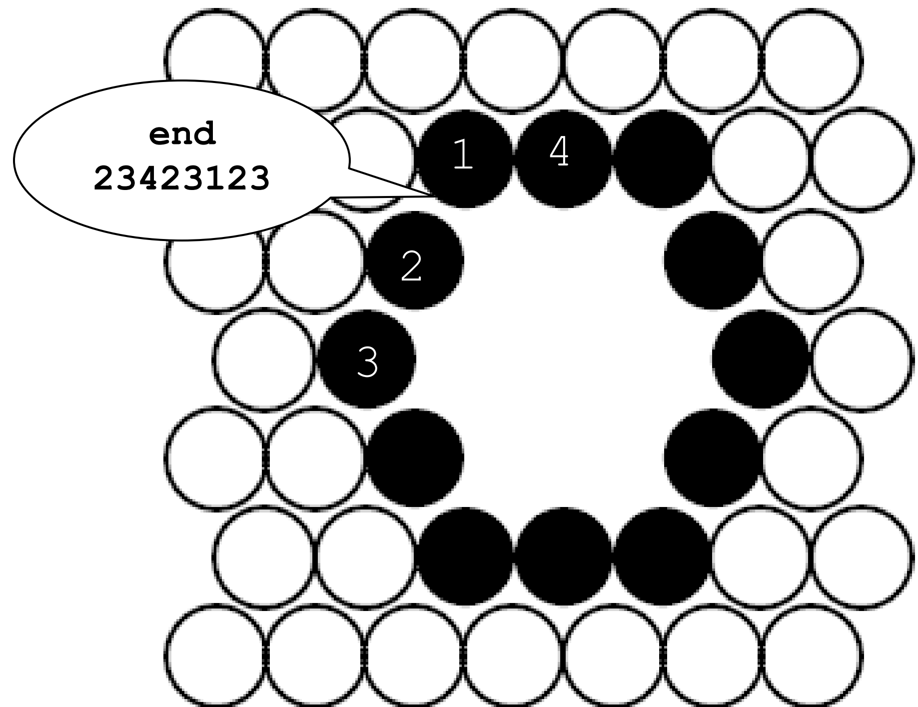
```
fail_block {
  // catom 1
  lnode->setBoss(this);
  rnode->setBoss(this);
}
...
setBoss(catom h) {
  myLeader = h;
}
```



Group Members: {1,2,4}

Op ID: 23423123

Failure Detect



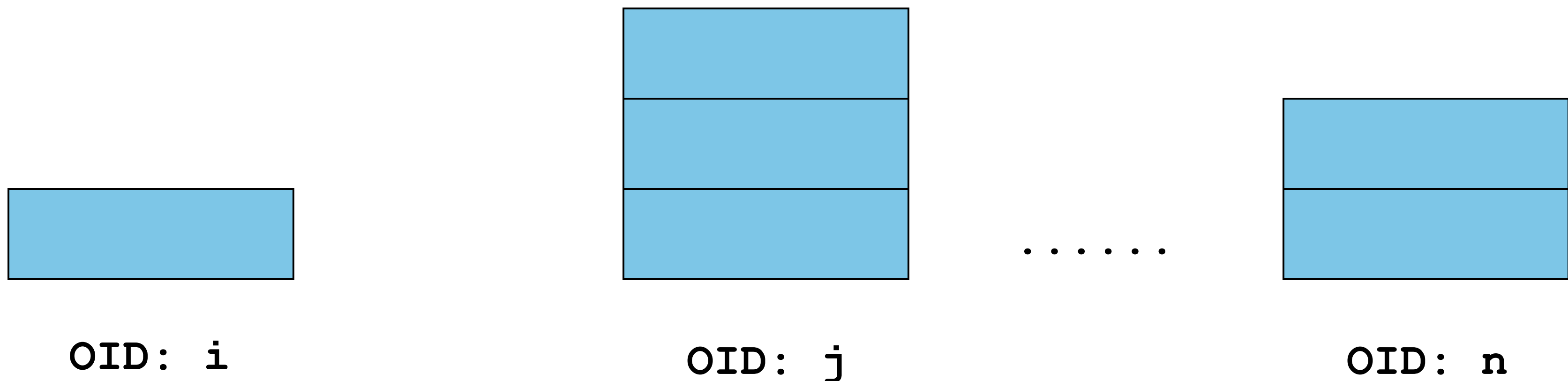
The `push_comp` Primitive

- `push_comp` ***b***
- On whichever catom it is called:
  - Suspend code in block *b*.
  - This code will be evaluated (purely for its side-effects) in the event that a failure is detected.
  - Called ‘compensating actions*’ or ‘compensations.’

*Westley Weimer and George C. Necula. Finding and preventing runtime error handling mistakes. In *OOPSLA ’04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA, 2004. ACM Press.

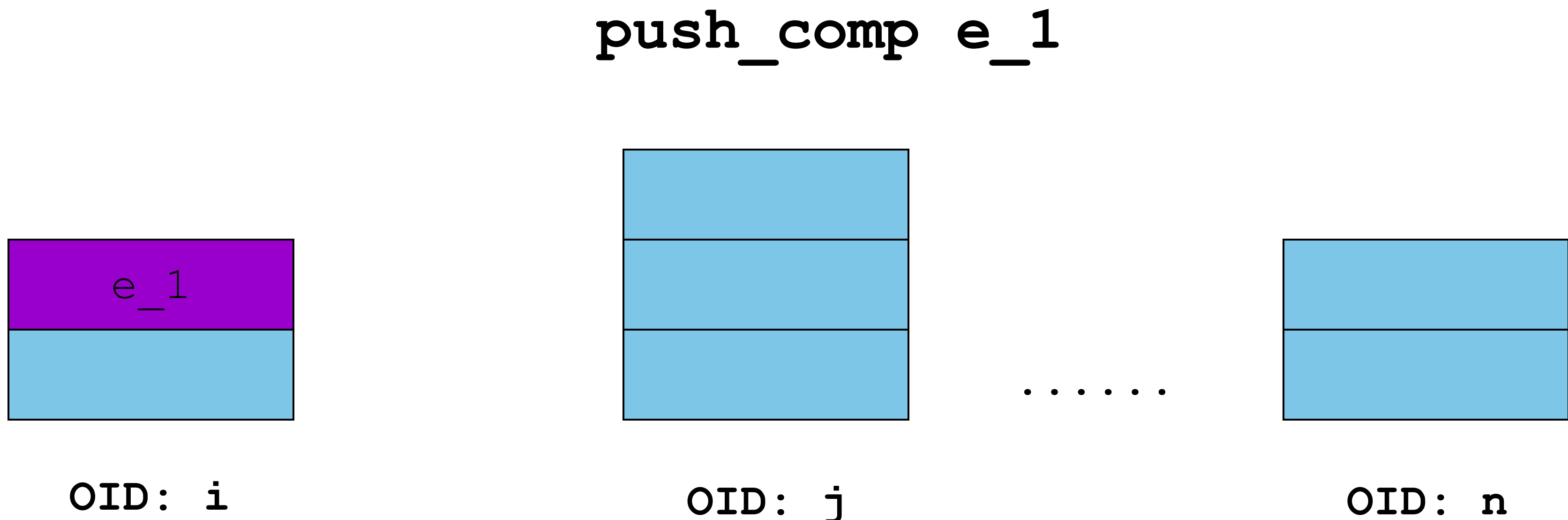
The `push_comp` Primitive

- Each catom has several stacks of compensations, one for each OID, and compensating actions are executed from top to bottom.



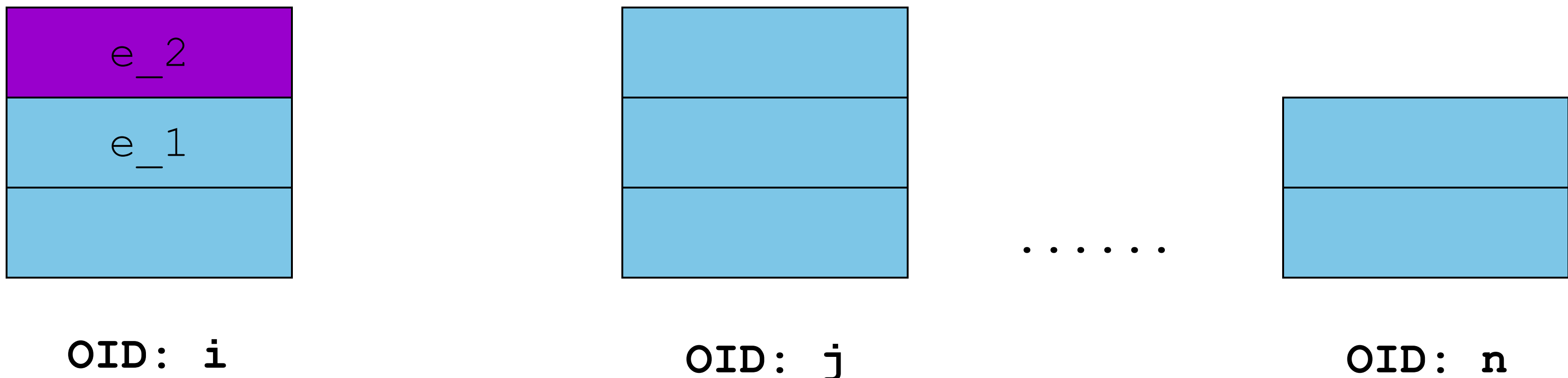
The push_comp Primitive

- Each catom has several stacks of compensations, one for each OID, and compensating actions are executed from top to bottom.



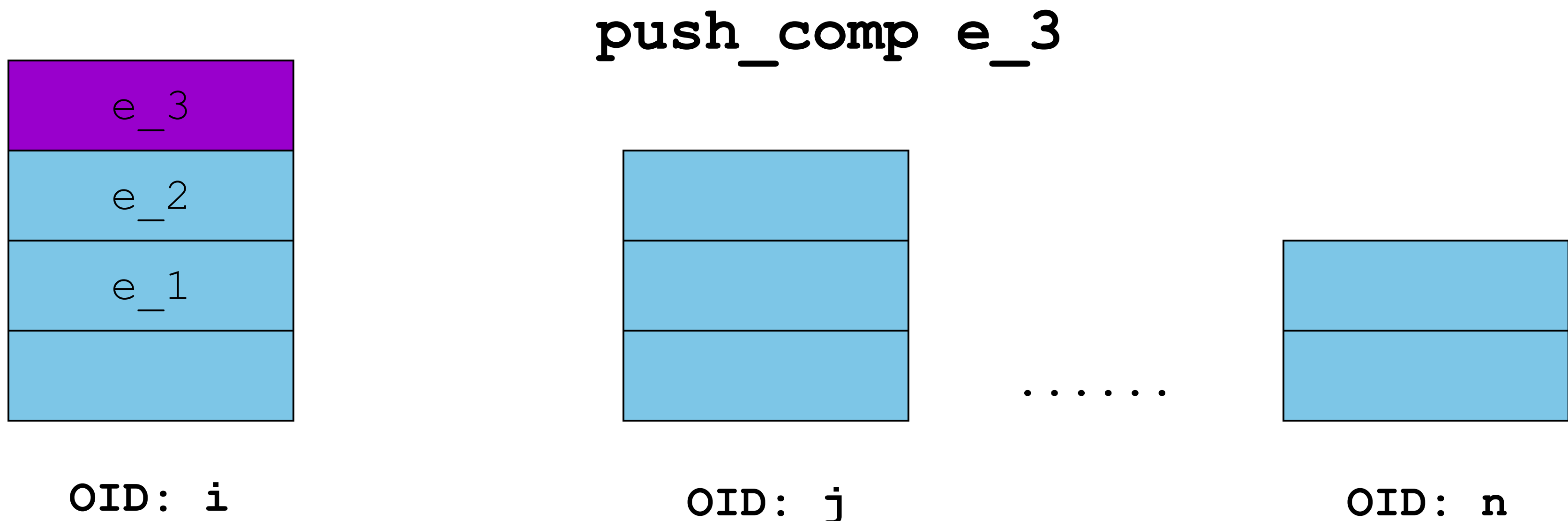
The `push_comp` Primitive

- Each catom has several stacks of compensations, one for each OID, and compensating actions are executed from top to bottom.



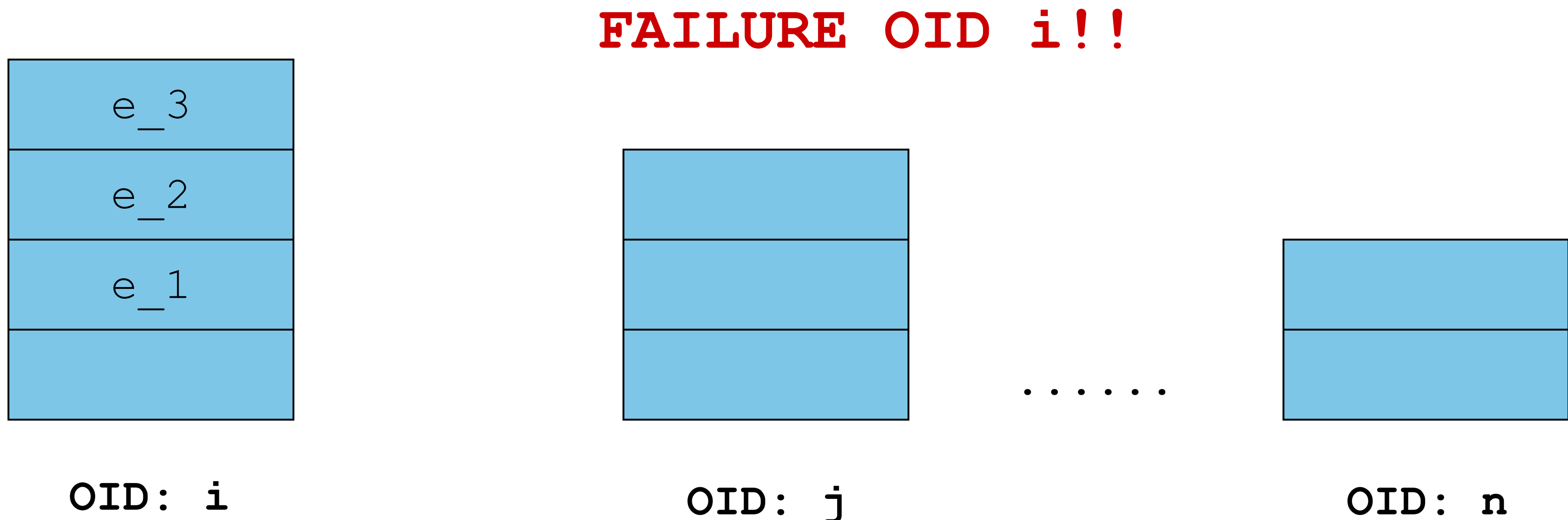
The `push_comp` Primitive

- Each catom has several stacks of compensations, one for each OID, and compensating actions are executed from top to bottom.



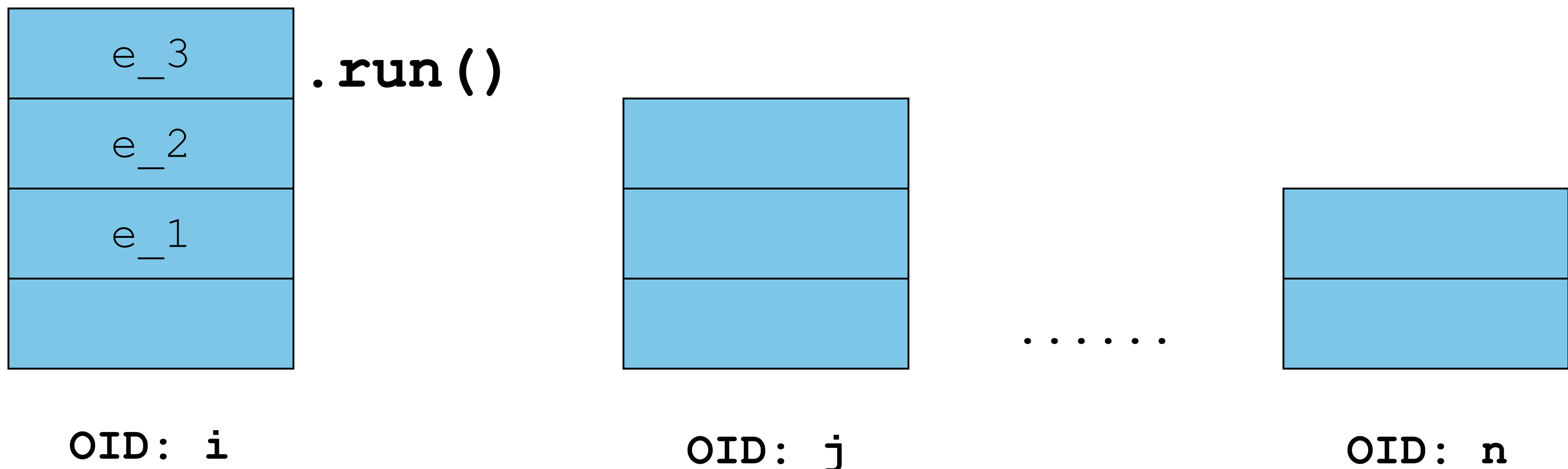
The `push_comp` Primitive

- Each catom has several stacks of compensations, one for each OID, and compensating actions are executed from top to bottom.



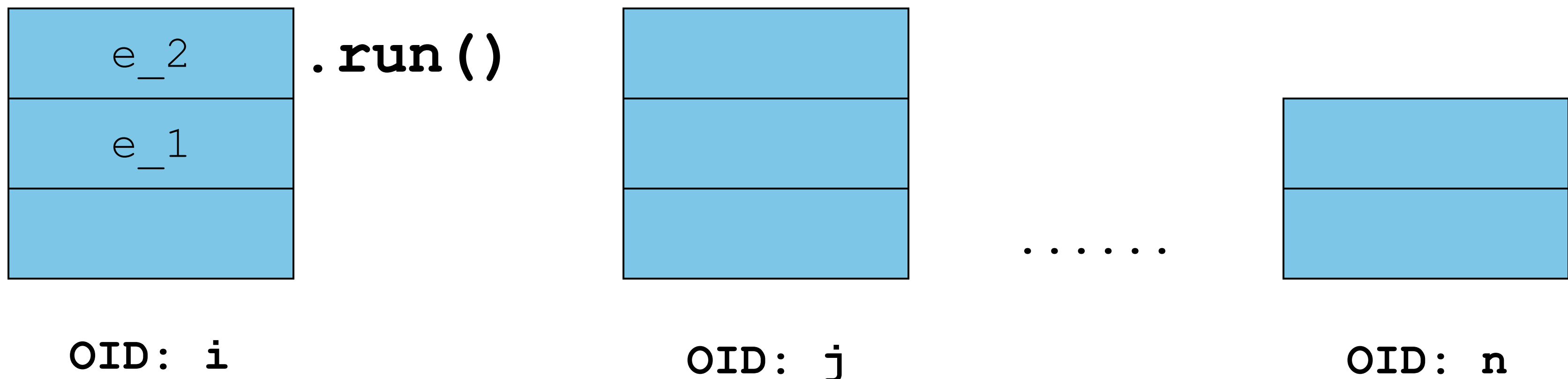
The `push_comp` Primitive

- Each catom has several stacks of compensations, one for each OID, and compensating actions are executed from top to bottom.



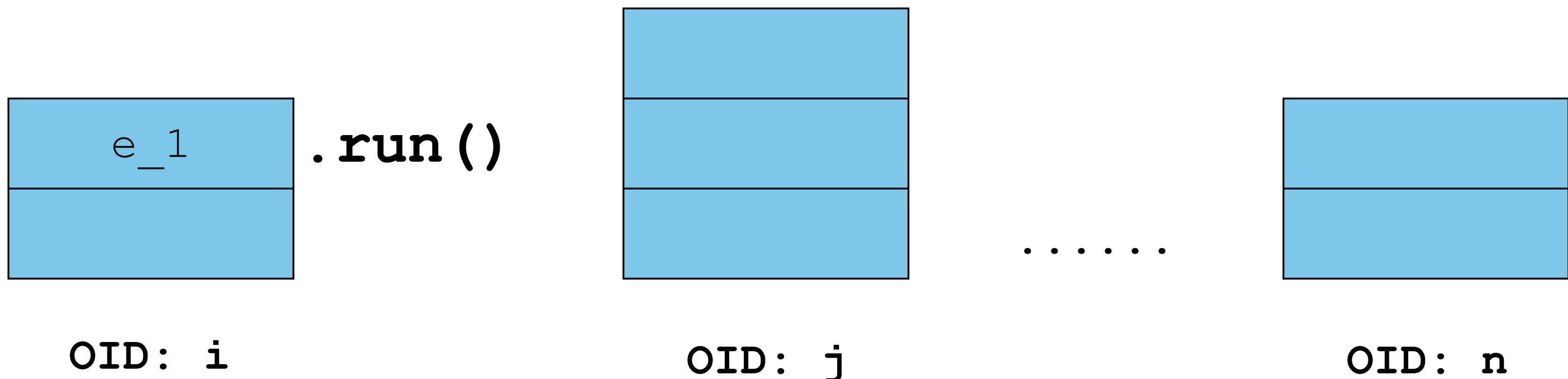
The `push_comp` Primitive

- Each catom has several stacks of compensations, one for each OID, and compensating actions are executed from top to bottom.



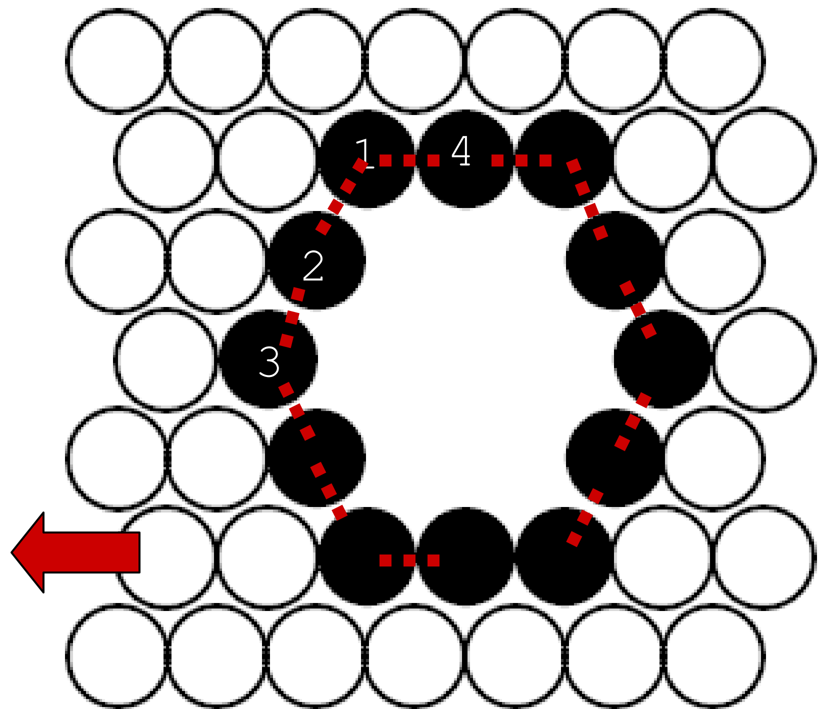
The `push_comp` Primitive

- Each catom has several stacks of compensations, one for each OID, and compensating actions are executed from top to bottom.



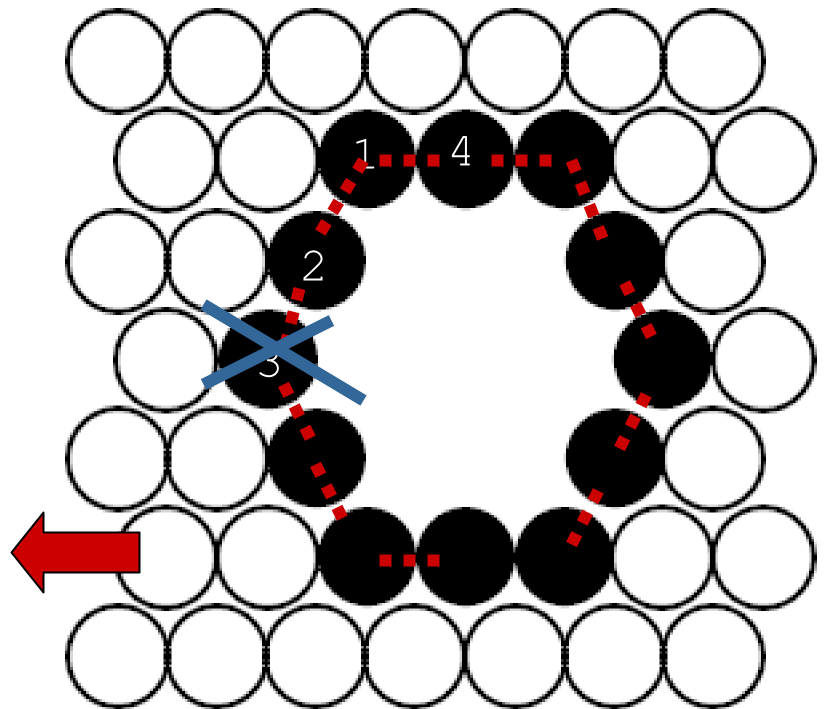
‘Demo,’ Continued

```
fail_block {
  // catom 1
  lnode->recurse(this,LEFT);
  rnode->recurse(this,RIGH);
}
recurse(catom ldr,Dir d){
  myLeader = ldr;
  push_comp(
     myLeader = -1);
  ...
  nnode->recurse(lead);
}
```



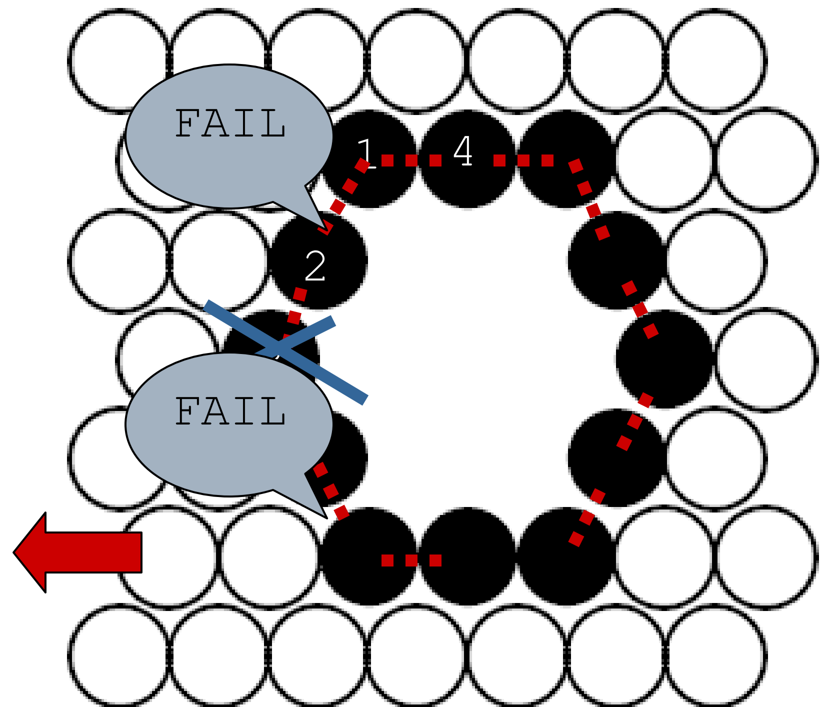
‘Demo,’ Continued

```
fail_block {
  // catom 1
  lnode->recurse(this,LEFT);
  rnode->recurse(this,RIGH);
}
recurse(catom ldr,Dir d){
  myLeader = ldr;
  push_comp(
     myLeader = -1);
  ...
  nnode->recurse(lead);
}
```



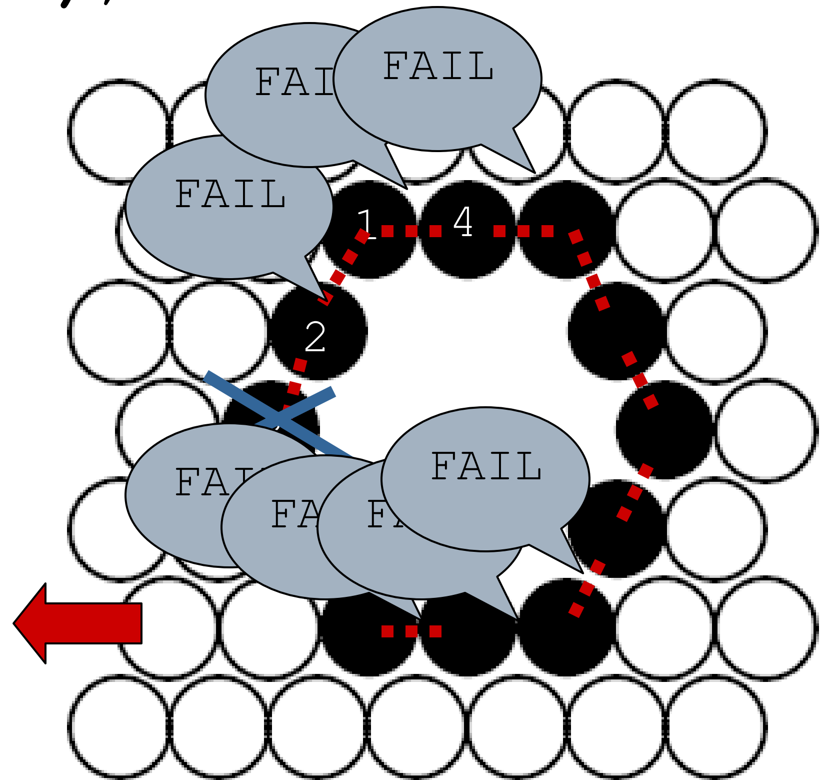
‘Demo,’ Continued

```
fail_block {
  // catom 1
  lnode->recurse(this,LEFT);
  rnode->recurse(this,RIGH);
}
recurse(catom ldr,Dir d){
  myLeader = ldr;
  push_comp(
     myLeader = -1);
  ...
  nnode->recurse(lead);
}
```



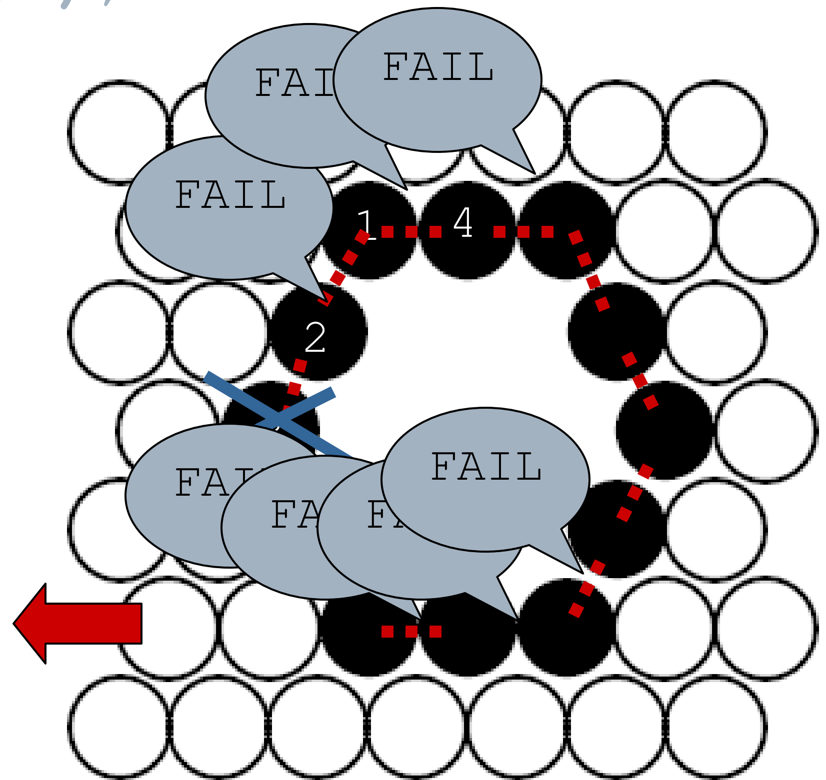
'Demo,' Continued

```
fail_block {
  // catom 1
  lnode->recurse(this,LEFT);
  rnode->recurse(this,RIGH);
}
recurse(catom ldr,Dir d){
  myLeader = ldr;
  push_comp(
     myLeader = -1);
  ...
  nnode->recurse(lead);
}
```



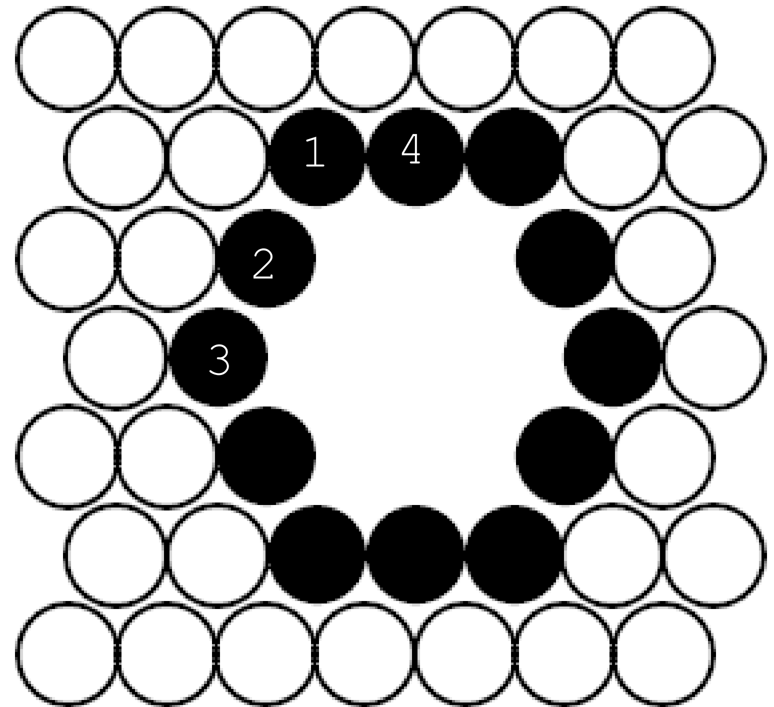
‘Demo,’ Continued

```
fail_block {
  // catom 1
  lnode->recurse(this,LEFT);
  rnode->recurse(this,RIGH);
}
recurse(catom ldr,Dir d){
  myLeader = ldr;
  push_comp(
     myLeader = -1);
  ...
  nnode->recurse(lead);
}
```



'Demo,' Continued

```
try {
fail_block {
  // catom 1
  lnode->recurse(this,LEFT);
  rnode->recurse(this,RIGH);
} } catch(OpFailure) {...
recurse(catom ldr,Dir d){
  myLeader = ldr;
  push_comp(
     myLeader = -1);
  ...
  nnode->recurse(lead);
}
```



Conclusion

- Failure Blocks
  - An extension to RPC for recovering from node failures.
  - Within a failure block
    - RPC calls add the callee to the current operation.
    - Callee and caller detect failure in one another.
    - Compensating actions can be stored, executed in the event of failure.
  - Targeted at modular robotic systems where failure is high but availability is important.

References

N. Beckman and J. Aldrich. A Programming Model for Failure-Prone, Collaborative Robots. To appear in the *2nd International Workshop on Software Development and Integration in Robotics (SDIR)*. Rome, Italy. April 14, 2007.

De Rosa, Goldstein, Lee, Campbell, Pillai. Scalable Shape Sculpting Via Hole Motion: Motion Planning in Lattice-Constrained Modular Robots. *IEEE ICRA 2006*. May 2006.

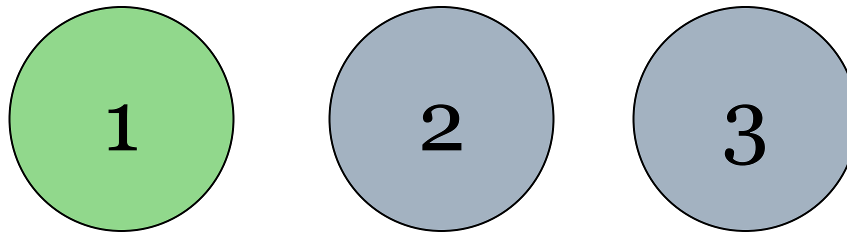
Achour Mostefaoui, Eric Mourgaya, and Michel Raynal. Asynchronous implementation of failure detectors. In *2003 International Conference on Dependable Systems and Networks (DSN'03)*, page 351, 2003.

Westley Weimer and George C. Necula. Finding and preventing runtime error handling mistakes. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA, 2004. ACM Press.

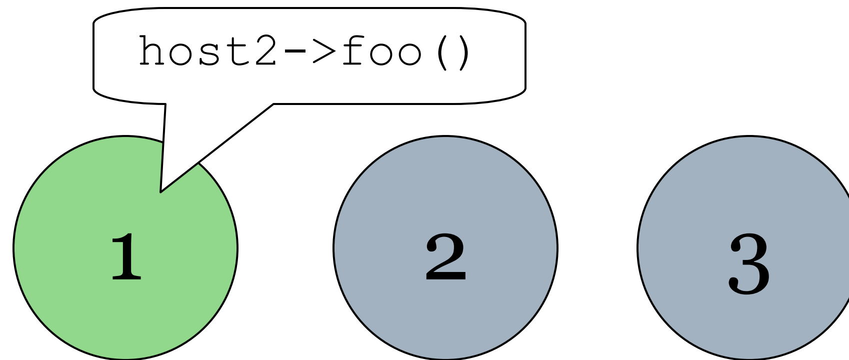
Michel Reynal. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36(1):53–70, 2005.

THE END

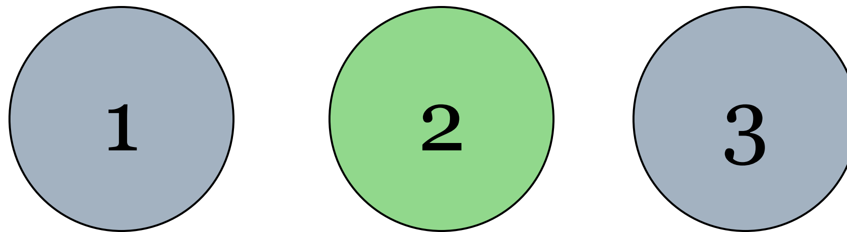
Scenario One



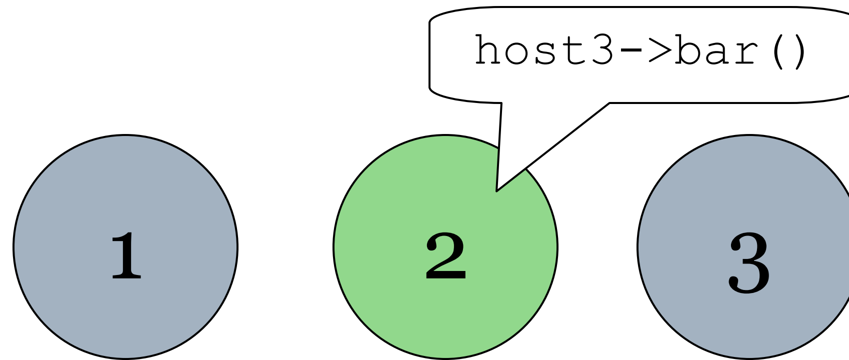
Scenario One



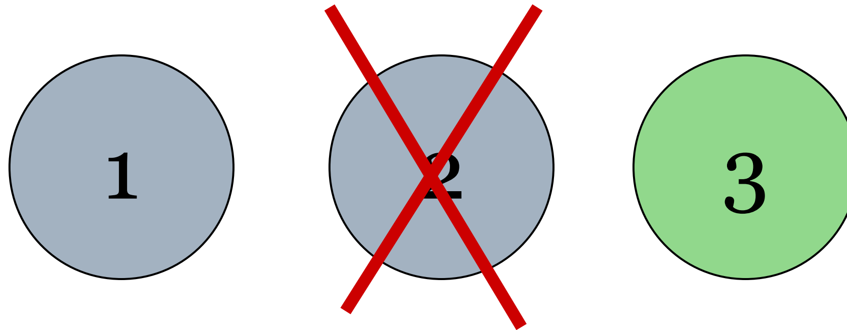
Scenario One



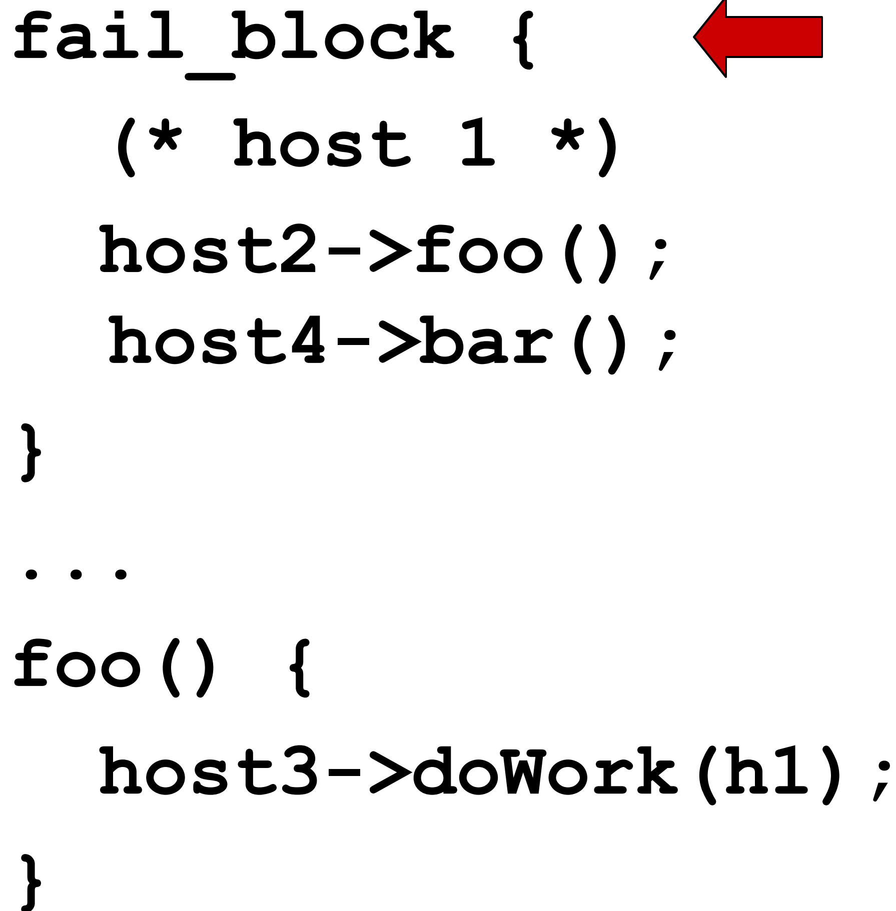
Scenario One



Scenario One

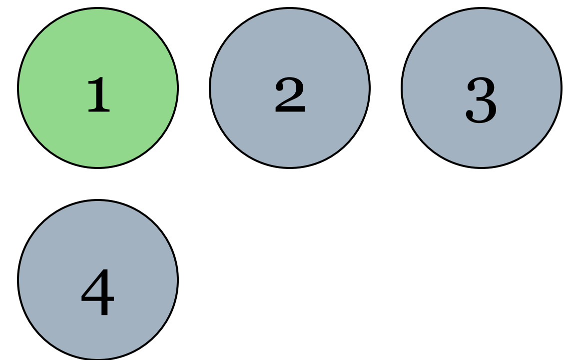


‘Demo’



Group Members: {}

Op ID:



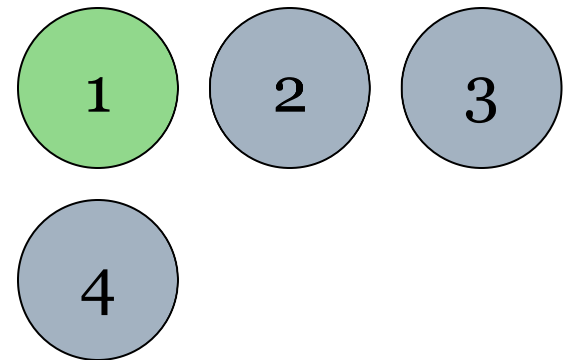
‘Demo’

```
fail_block {
  (* host 1 *)
  host2->foo();
  host4->bar();
}
...
foo() {
  host3->doWork(h1);
}
```

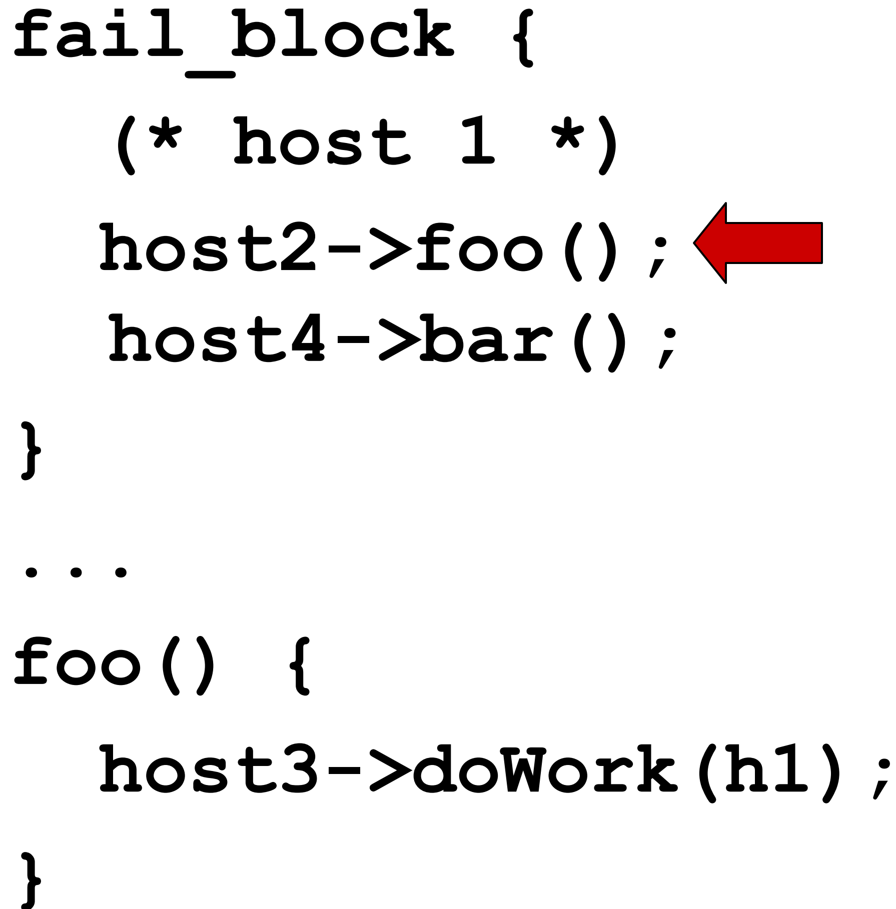
Group Members: {1}

Op ID: 3435435

Regular Ping

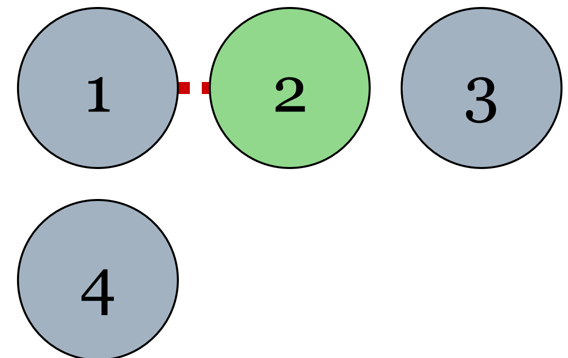


‘Demo’



Group Members: {1,2}

Op ID: 3435435



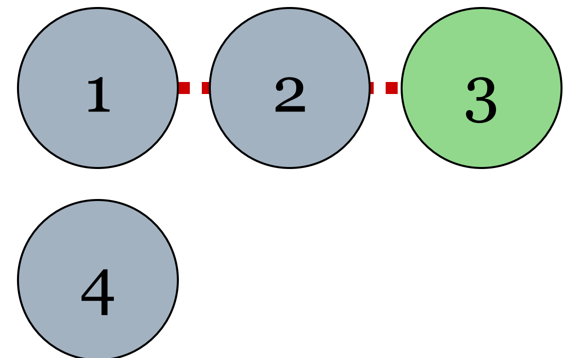
‘Demo’

```
fail_block {
  (* host 1 *)
  host2->foo();
  host4->bar();
}
...
foo() {
  host3->doWork(h1);
}
```

Group Members: {1,2,3}

Op ID: 3435435

Regular Ping



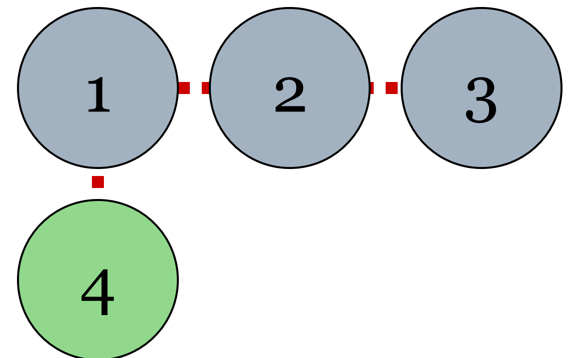
‘Demo’

```
fail_block {
  (* host 1 *)
  host2->foo();
  host4->bar();
}
...
foo() {
  host3->doWork(h1);
}
```

Group Members: {1,2,3,4}

Op ID: 3435435

Regular Ping



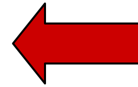
'Demo'

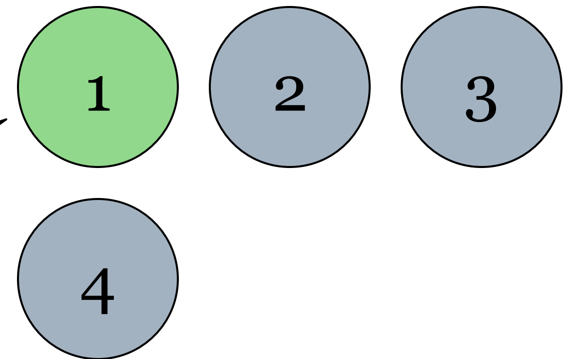
```
fail_block {
  (* host 1 *)
  host2->foo();
  host4->bar();
}

...

foo() {
  host3->doWork(h1);
}
```

Group Members: {}

Op ID:

Regular Ping

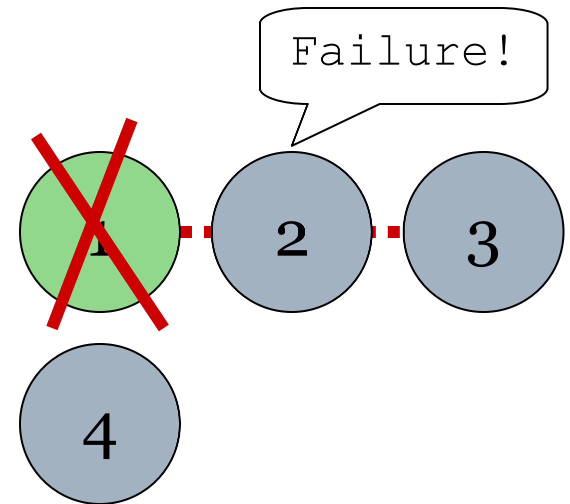


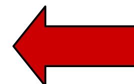


At a Macroscopic Level… (Video)

‘Demo,’ Continued

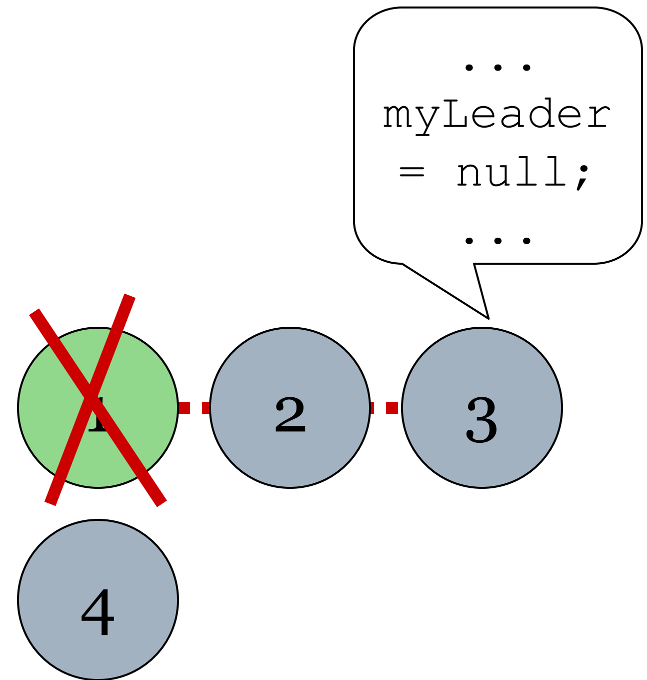
```
...
doWork(HostAddr a) {
  myLeader = a;

  push_comp {
    if(myLeader == a)
      myLeader = null;
  }
}
...
```

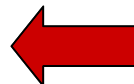




'Demo,' Continued

```
...
doWork(HostAddr a) {
  myLeader = a;

  push_comp {
    if(myLeader == a)
      myLeader = null;
  }
}
...
```





Outline

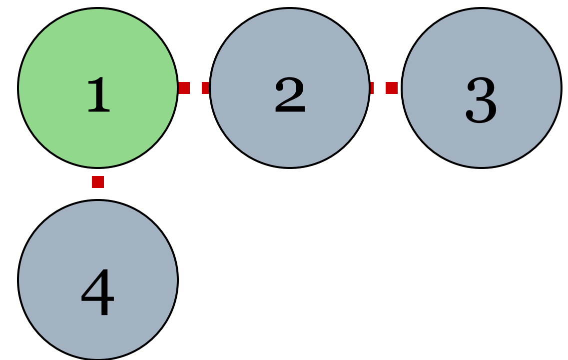
- The Rate of Failure Will be High
- Two Failure Scenarios We Would Like to Handle
- Existing RPC Systems Do Not Meet Our Needs
- Our Model Has Two Key Pieces
    - `fail_block`
    - `push_comp`
- Our Model Does Not Require Consistency

Our System Does Not Require Consistency

- Our model has a nice feature:
  - We do not require consistency in failure detection!
  - This has been proven to be impossible in 'time-free' systems.

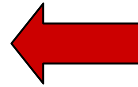
What is Consistency?

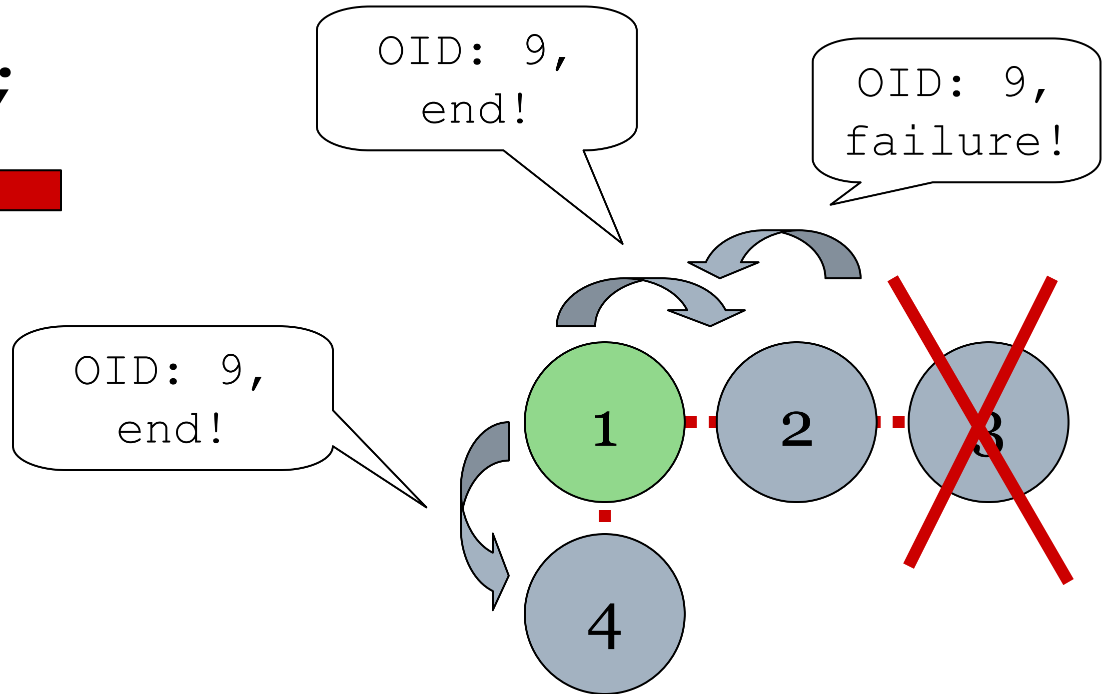
```
fail_block {
   (* host 1 *)
  host2->foo();
   host4->bar();
}
...
foo() {
  host3->doWork(h1);
}
```



What is Consistency?

```
fail_block {
  (* host 1 *)
  host2->foo();
  host4->bar();
}
...
foo() {
  host3-
  >doWork(h1);
}
```





Our System Does Not Require Consistency

- Domain Assumption:
  - The ultimate goal of any application is to perform actuator movements.
- Additionally,
  - The thread of control must migrate to a catom in order to issue an actuator command.
  - If a thread migrates to a catom that has detected or knows about a failure, that thread will not continue normally.

Our System Does Not Require Consistency

- Therefore, if inconsistency occurs, we know:
  - In between detection and `fail_block` completion, no actuator movements were necessary on any hosts that knew about the failure.
  - In the sense that actuator movements are the ultimate goal in the domain, their work was already done.

Our System Does Not Require Consistency

- What if we won't make an actuator movement on a host, but we need to know it performed its duty?
  - E.g., structural catoms
- This is a question of live-ness versus other goals.
  - `fail_block` should be used precisely when live-ness is a chief concern.

Assumptions

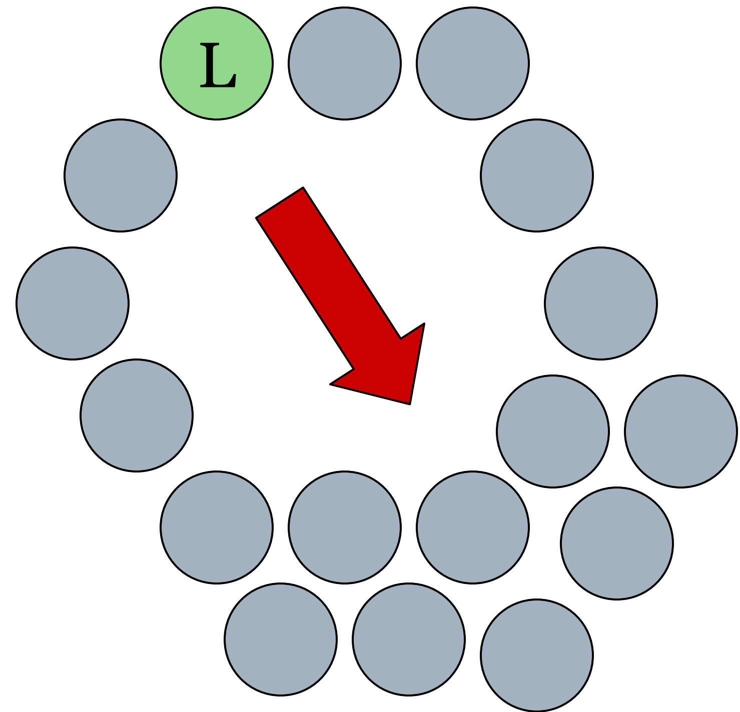
- Movement:
  - When a movement occurs, you are required to talk with these surrounding hosts and they will be able to figure out the new location to ping.
- Goals:
  - Actuator movements are the ultimate goal of most applications in this domain.

What about Transactions?

- Semantics of roll-back suggest a transactional model.
- Similarly, it seems that Two-Phase commit could give us consistency.
- But
  - 2PC has one or two extra rounds of communication
    - Application doesn't make progress!
    - Our model has no extra blocking rounds.
  - 2PC can block indefinitely if the coordinator fails
    - In non-blocking protocols the number of failures is bounded.
  - It is not clear how error detection and 2PC could be combined.

In Detail…

- But, after this field has been set, failure of the leader leaves the catoms in a dead state.



The `push_comp` Primitive

- We call this suspended code ‘compensating actions.’
  - Borrowed terminology from Weimar and Necula.
  - Originally used to ensure proper clean-up for file handlers, etc. in exceptional circumstances.
    - (However, our compensating actions are only executed when a failure is detected.)

Why Server-Side Failure Handling?

- The client may *think* the server has failed, when it hasn't.
  - Allow server to return to a stable state.
  - Failure detectors unreliable in 'time-free' systems.
- The client may have failed.
- An catom on the return route may have failed.