

HOW TO MAKE CHORD CORRECT

Pamela Zave

AT&T Laboratories—Research

Florham Park, New Jersey, USA

CHORD IS A DISTRIBUTED HASH TABLE:

AN AD-HOC

PEER-TO-PEER NETWORK

IMPLEMENTING A

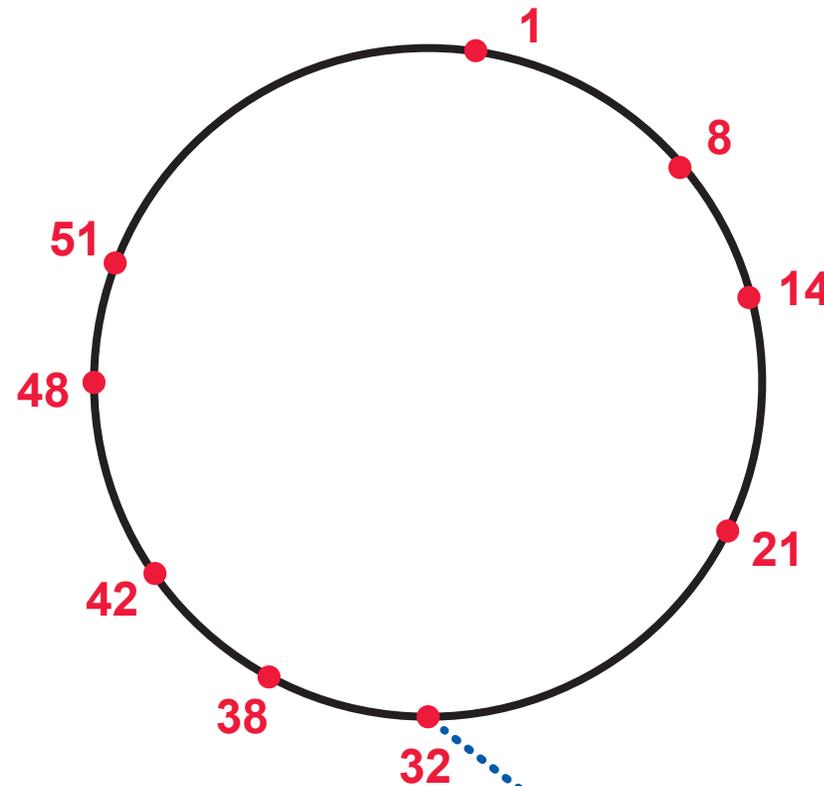
KEY-VALUE STORE

identifier of a node (assumed unique) is an m -bit hash of its IP address

$m = 6$

keys are also m bits

nodes are arranged in a ring, each node having a successor pointer to the next node (in integer order with wraparound at 0)



storage and lookup rely on the ring structure

key-value pairs for keys 22 - 32 are stored here

the ring-maintenance protocol preserves the ring structure as nodes join and leave silently or fail

WHY IS CHORD IMPORTANT?

*the SIGCOMM paper introducing Chord
is the 4th-most-referenced
paper in computer science, . . .*

. . . and won SIGCOMM's 2011 Test of Time Award

APPLICATIONS OF DISTRIBUTED HASH TABLES

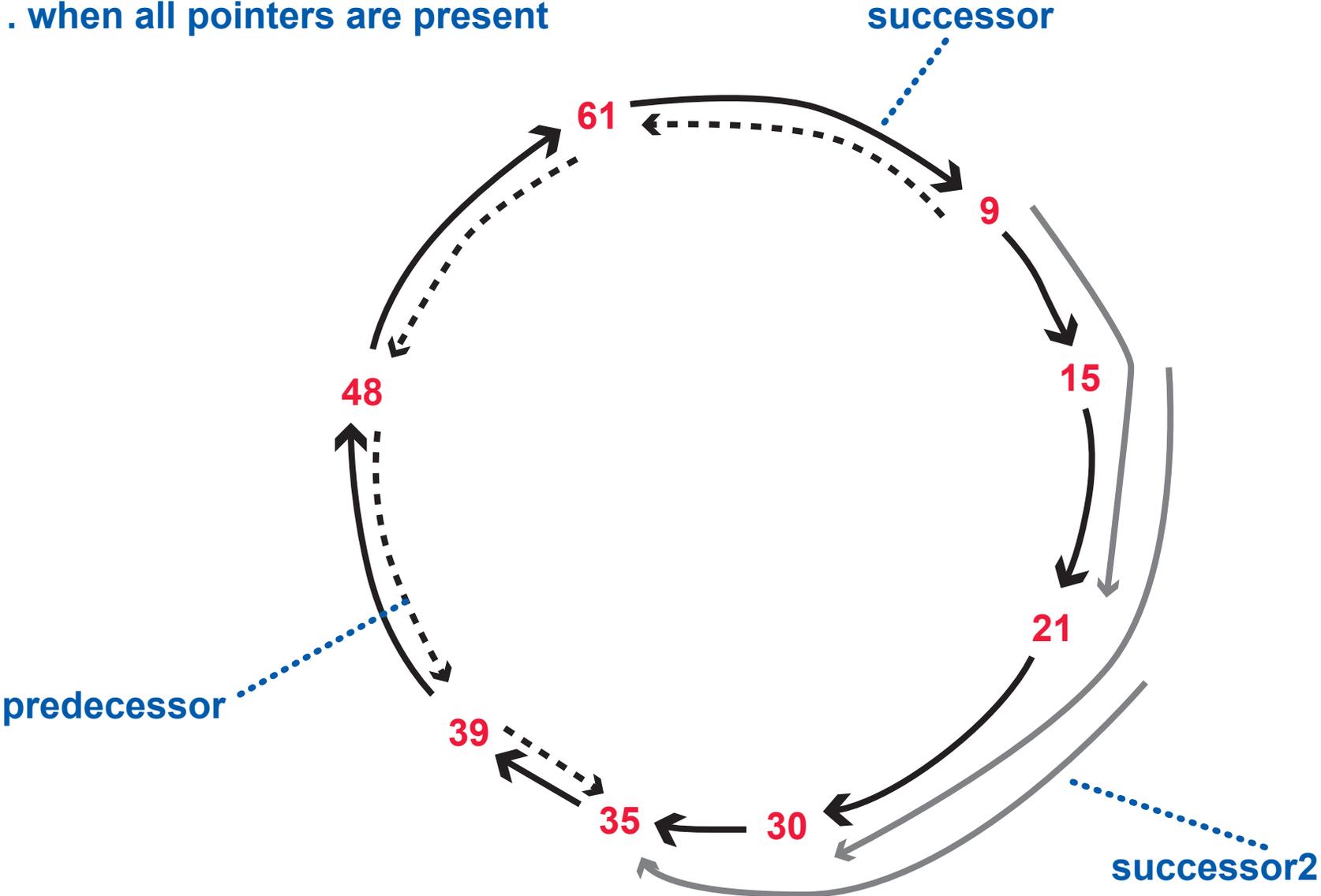
- allow millions of peers to cooperate in implementing a data store
- used as a building-block in fault-tolerant applications
- the best-known application is BitTorrent

OTHER DISTRIBUTED HASH TABLES

- Pastry
- Tapestry
- CAN
- Kademlia
- and others

AN IDEAL NETWORK . . .

. . . when all pointers are present

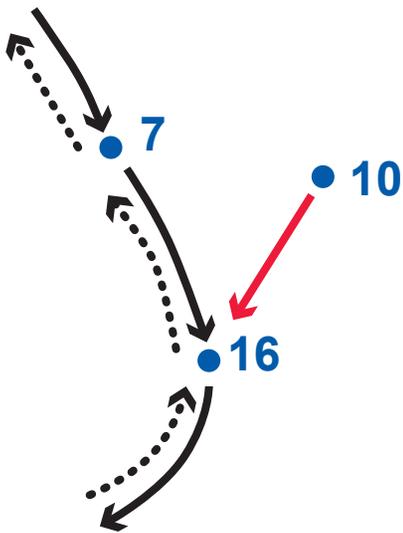


OPERATIONS OF THE RING-MAINTENANCE PROTOCOL

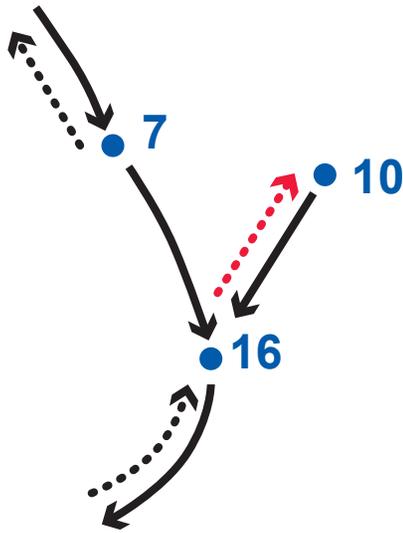
an operation changes the state of one node

most operations are scheduled, asynchronously and autonomously, by their own nodes

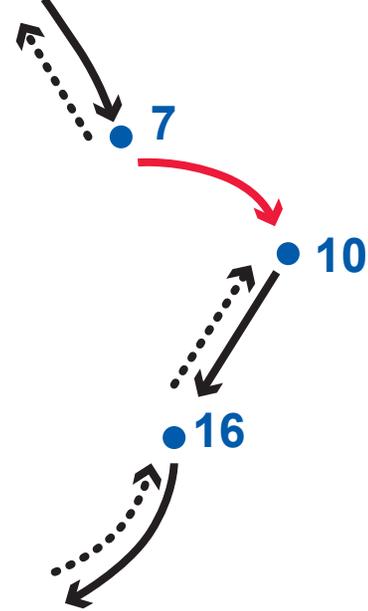
10 JOINS



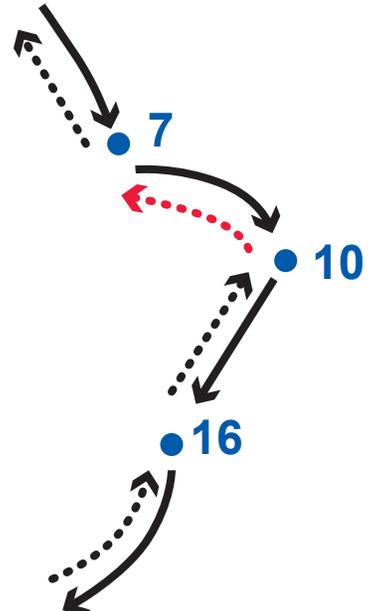
16 NOTIFIED



7 STABILIZES



10 NOTIFIED

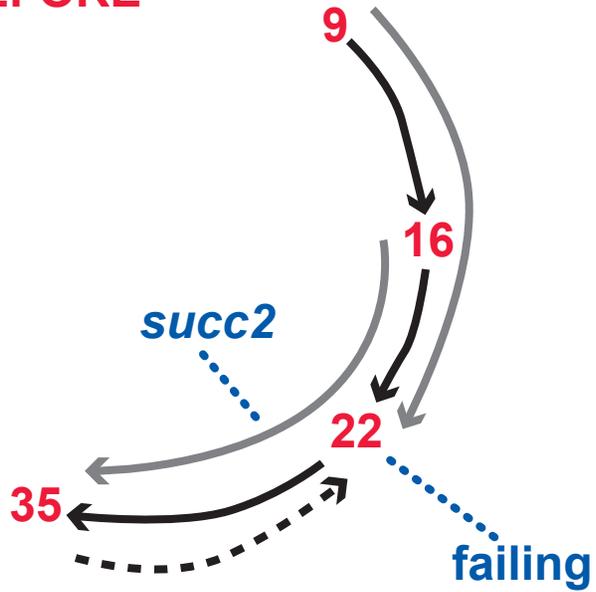


just as Stabilize and Notified operations repair the disruption caused by Joins, . . .

. . . Update, Reconcile, and Flush operations repair the disruption caused by Failures (using redundant successors)

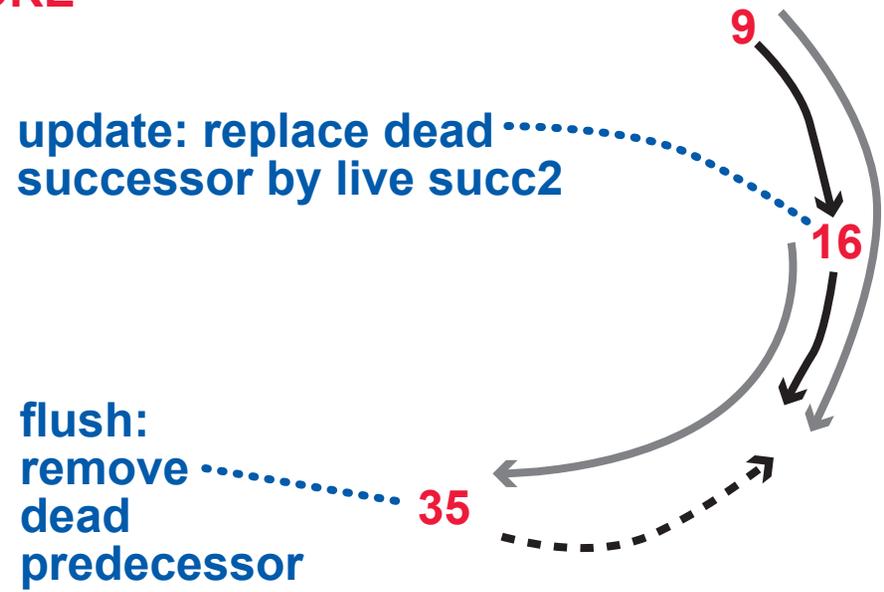
A FAILURE . . .

BEFORE

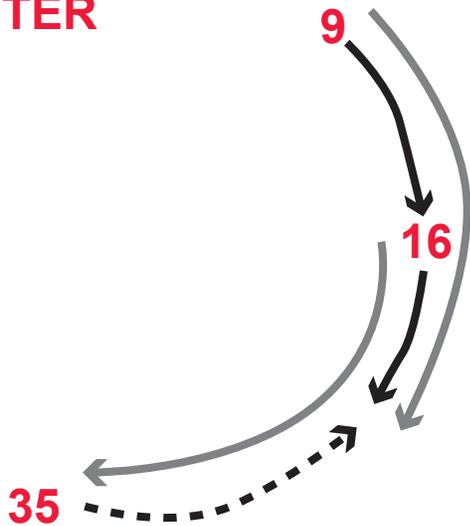


. . . AND ITS REPAIR

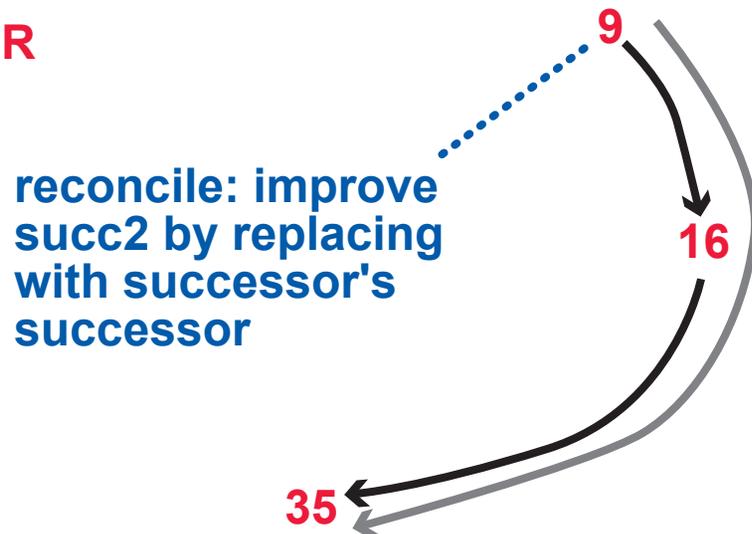
BEFORE



AFTER



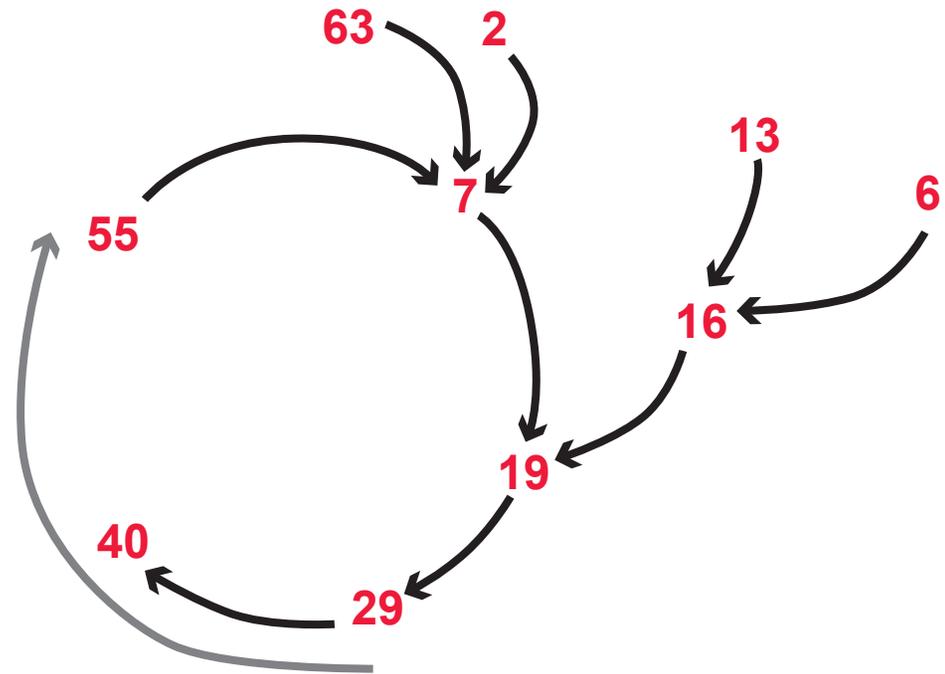
AFTER



A VALID NETWORK

defining a node's *best successor* as its first successor pointing to a live node (member):

- there is a cycle of best successors
- there is no more than one cycle
- on the cycle of best successors, the nodes are in identifier order
- from each member not in the cycle, the cycle is reachable through best successors



WHAT THE PROTOCOL CAN DO (allegedly)

- keep the network valid at all times
- repair any other defect (appendages, missing pointers, etc.) . . .
... so that eventually, if there are no new joins or failures, the network becomes ideal
- there are no intervals in which sets of nodes are “locked” to implement multi-node atomic operations

great performance! fast and easy to analyze!

WHAT THE PROTOCOL CANNOT DO

if the network becomes invalid, the protocol cannot repair it

THE CLAIMS

"Three features that distinguish Chord from many peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance."

THE REALITY

- even with simple bugs fixed and optimistic assumptions about atomicity, the original protocol is not correct
- of the seven properties claimed invariant of the original version, not one is actually an invariant
- some (or maybe all) of the many papers analyzing Chord performance are based on false assumptions about how the protocol works

DO REAL IMPLEMENTATIONS HAVE THESE FLAWS?

- some implementations have even the easiest-to-fix flaws
- almost certain that all implementations have some flaws
- cannot tell for sure without reading the code, as implementors do not document what they have actually implemented

THE GOAL

- find a specification that is actually correct
- persuade people to take the specification seriously

LIGHTWEIGHT MODELING

DEFINITION

- constructing a small, abstract logical model of the key concepts of a system
- analyzing the properties of the model with a tool that performs exhaustive enumeration over a bounded domain

WHY IS IT "LIGHTWEIGHT"?

- because the model is very abstract in comparison to a real implementation, it is small and can be constructed quickly
- because the analysis tool is "push-button", it yields results with relatively little effort

*in contrast,
theorem proving is not "push-button"*

WHY IS IT INTERESTING?

- it is a proven tool for revealing conceptual errors and improving software quality, in a cost-effective manner

*you will see how little work
it takes to find problems
with Chord*

- it is a formal method that can be used and appreciated by very practical people

*protocol designers
should model as they design*

- it is easy (at least to get started) and fun!

*"If you like surprises, you will
love lightweight modeling."*

—Pamela Zave

MY FAVORITE TOOLS

Promela (language) / Spin

- Promela is a simple programming language with concurrent processes, messages, bounded message queues, and fixed-size arrays.
- Spin is a model checker: the program specifies a large finite-state machine which the checker explores exhaustively.

Alloy (language) / Alloy Analyzer

- Alloy combines relational algebra, first-order predicate calculus, transitive closure, and objects.
- Analyzer compiles a model into a set of Boolean constraints, uses SAT solvers to decide whether the set of constraints is satisfiable.

*the style of modeling in these two languages
is radically different*

the analysis capabilities are also radically different

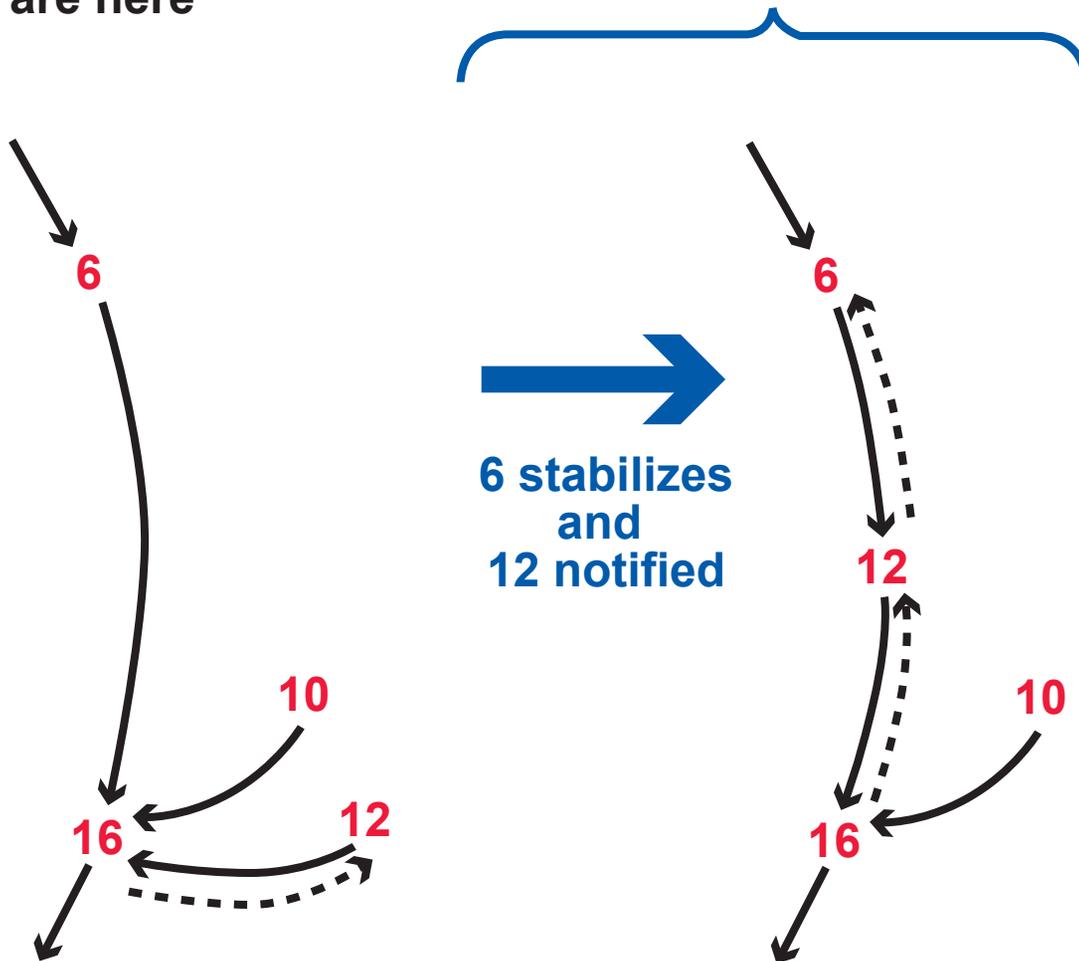
*both are applicable to Chord
(see “A practical comparison of Alloy and Spin”)
but this talk uses Alloy*

A PROPERTY CLAIMED INVARIANT

OrderedMerges . . .

. . . means that appendages are in the correct places, as they are here

this property is easily violated, as shown here



The good news:

- violations are repaired by stabilization

The bad news:

- causes some lookups to fail
- invalidates some assumptions used in performance analysis

The main point:

How could this go unknown for ten years?

- behavior appears in networks with 3 nodes
- it takes an 88-line model and .3 seconds of analysis to find this with Alloy

RELATIONAL JOIN

THE KEY TO UNDERSTANDING RELATIONAL ALGEBRA (AND ALLOY)

RELATIONS

P is of type A

Q is of type A -> B -> C

R is of type C -> D

A\$1

A\$0 -> B\$0 -> C\$0

C\$0 -> D\$0

A\$1 -> B\$1 -> C\$1

C\$1 -> D\$1

A\$2

A\$2 -> B\$2 -> C\$2

JOIN EXPRESSION

P . Q . R columns on either side of dot must have same type

COMPUTATION OF JOIN

value in “shared column” must match

A\$1

~~A\$0 -> B\$0 -> C\$0~~

C\$0 -> D\$0

A\$1 -> B\$1 -> C\$1

C\$1 -> D\$1

A\$2

A\$2 -> B\$2 -> C\$2

in resulting relation, “shared columns” are removed

VALUE OF JOIN EXPRESSION

B\$1 -> D\$1

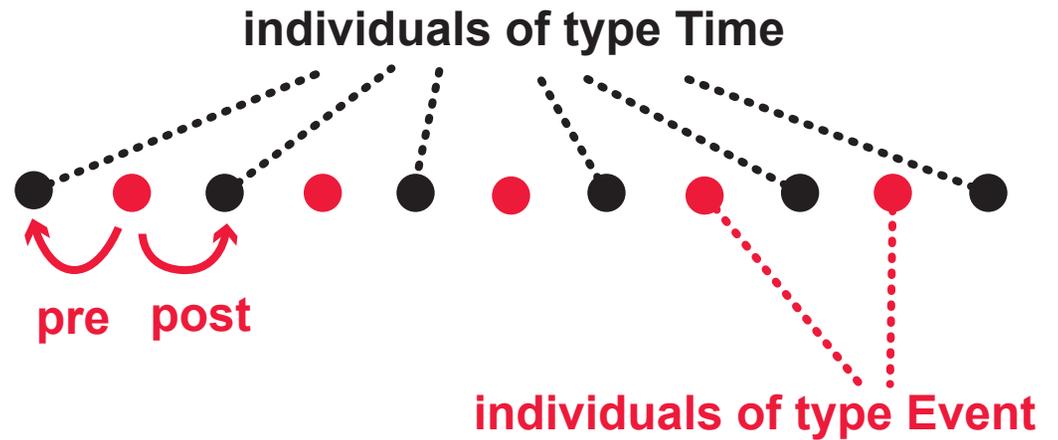
result is a relation with any number of tuples, including zero or many

TIME IN ALLOY: PART OF THE MODEL YOU WRITE, NOT PART OF THE LANGUAGE YOU WRITE IN

sig Time {
..... a basic type, declared
to be totally ordered

sig Event {
..... an object type,
with two fields
pre: Time,
post: Time }

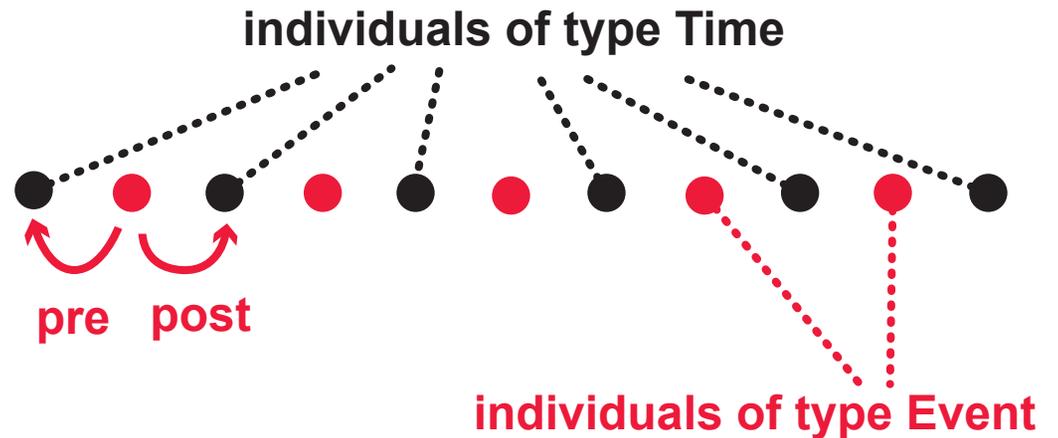
Alloy “facts” produce
these relationships



TIME IN ALLOY: PART OF THE MODEL YOU WRITE, NOT PART OF THE LANGUAGE YOU WRITE IN

```
sig Time { }
```

```
sig Event {  
  pre: Time,  
  post: Time }  
  an object type,  
  with two fields
```



OBJECTS IN ALLOY HAVE A FUNDAMENTALLY SIMPLE RELATIONAL SEMANTICS

pre is a relation from Event to Time . . .

```
Event$0 -> Time$0  
Event$1 -> Time$1  
...
```

. . . so if e stands for Event\$1,
then e . pre is Time\$1

TEMPORAL STATE IN ALLOY

sig Node {

 succ: Node lone -> Time,

 prdc: Node lone -> Time }

 succ is a ternary relation from
 Node to Node to Time

 for each Node, each Time
 corresponds to one or zero
 predecessor Nodes

TEMPORAL STATE IN ALLOY

sig Node {

 succ: Node lone -> Time,

 prdc: Node lone -> Time }

{ all t: Time | no succ.t => no prdc.t }

if a Node is not a member of the network it has no successor . . .

. . . in which case it cannot have a predecessor, either;

stated separately from the signature it would look like this:

fact { all n: Node, t: Time |

 no n.succ.t => no n.prdc.t }

TEMPORAL STATE IN ALLOY

```
sig Node {
```

```
    succ: Node lone -> Time,
```

```
    prdc: Node lone -> Time }
```

```
{ all t: Time | no succ.t => no prdc.t }
```

Nodes are also declared to be totally ordered, so we can use library predicates to define cycle ordering:

```
pred Between [n1, n2, n3: Node] {
```

```
    lt [n1,n3]
```

```
    => ( lt [n1,n2] && lt [n2,n3] )
```

```
    else ( lt [n1,n2] || lt [n2,n3] )
```

```
}
```

special case for wraparound at zero



GRAPH PROPERTIES IN ALLOY

transitive closure

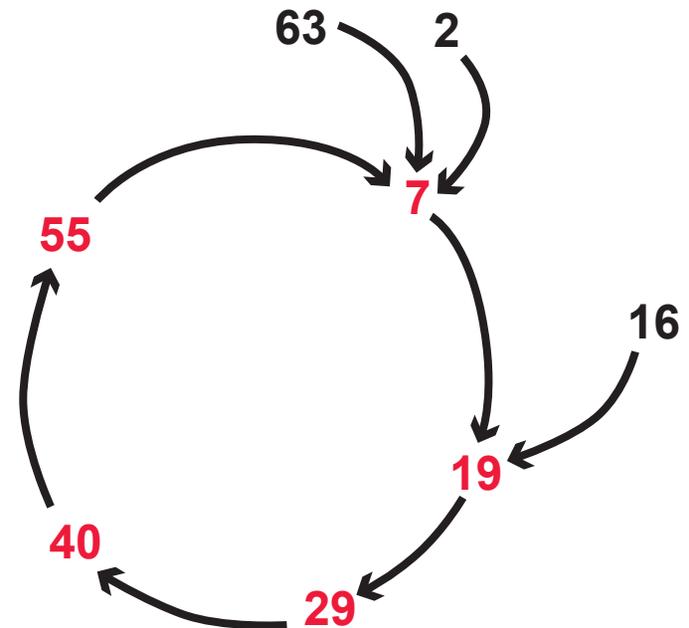
```
pred OneOrderedRing [t: Time] {
```

```
  let ringMembers = { n: Node | n in n.^(succ.t) } |
```

ringMembers is the set of all nodes that are members (because they have successors) . . .

. . . and that are reachable from themselves by following successor pointers

```
}
```



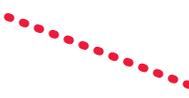
GRAPH PROPERTIES IN ALLOY

```
pred OneOrderedRing [t: Time] {
```

```
  let ringMembers = { n: Node | n in n.^(succ.t) } |
```

```
    some ringMembers
```

there is at least
one ring



```
}
```

GRAPH PROPERTIES IN ALLOY

```
pred OneOrderedRing [t: Time] {
```

```
  let ringMembers = { n: Node | n in n.^(succ.t) } |
```

```
    some ringMembers
```

```
    && (all disj n1, n2: ringMembers | n1 in n2.^(succ.t) )
```

```
}
```

there is at most
one ring



GRAPH PROPERTIES IN ALLOY

```
pred OneOrderedRing [t: Time] {
```

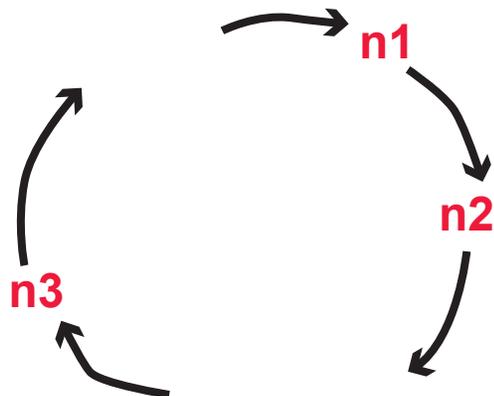
```
  let ringMembers = { n: Node | n in n.^(succ.t) } |
```

```
    some ringMembers
```

```
    && (all disj n1, n2: ringMembers | n1 in n2.^(succ.t)) )
```

```
    && (all disj n1, n2, n3: ringMembers | n2 = n1.succ.t => ! Between [n1,n3,n2] )
```

```
}
```



in the ring, nodes are ordered by identifier

EVENTS IN ALLOY

sig RingEvent extends Event { node: Node }

this subtype
adds a field



sig Stabilize extends RingEvent { }

fact StabilizeChangesSuccessor {

all s: Stabilize, n: s.node, t: s.pre |

shorthands



this fact will describe

Stabilize events

}

EVENTS IN ALLOY

sig RingEvent extends Event { node: Node }

sig Stabilize extends RingEvent { }

fact StabilizeChangesSuccessor {

all s: Stabilize, n: s.node, t: s.pre |

let newSucc = (n.succ.t).prdc.t |

{

this node's
successor

its predecessor

} }

using a shared-state model
of distributed computing,
newSucc is this node's
successor's predecessor

EVENTS IN ALLOY

```
sig RingEvent extends Event { node: Node }
```

```
sig Stabilize extends RingEvent { }
```

```
fact StabilizeChangesSuccessor {
```

```
  all s: Stabilize, n: s.node, t: s.pre |
```

```
    let newSucc = (n.succ.t).prdc.t |
```

```
    { some n.succ.t
```

```
      some newSucc
```

```
      Between[n,newSucc,n.succ.t]
```

preconditions:

this node is a member

newSucc exists

newSucc is a better successor

```
  } }
```

EVENTS IN ALLOY

```
sig RingEvent extends Event { node: Node }
```

```
sig Stabilize extends RingEvent { }
```

```
fact StabilizeChangesSuccessor {
```

```
  all s: Stabilize, n: s.node, t: s.pre |
```

```
    let newSucc = (n.succ.t).prdc.t |
```

```
    { some n.succ.t
```

```
      some newSucc
```

```
      Between[n,newSucc,n.succ.t]
```

```
      n.succ.(s.post) = newSucc .....postconditions: this node's successor  
                                             becomes newSucc
```

```
  } }
```

EVENTS IN ALLOY

```
sig RingEvent extends Event { node: Node }
```

```
sig Stabilize extends RingEvent { }
```

```
fact StabilizeChangesSuccessor {
```

```
  all s: Stabilize, n: s.node, t: s.pre |
```

```
    let newSucc = (n.succ.t).prdc.t |
```

```
    { some n.succ.t
```

```
      some newSucc
```

```
      Between[n,newSucc,n.succ.t]
```

```
      n.succ.(s.post) = newSucc
```

```
      ( all m: Node | m != n => m.succ.(s.post) = m.succ.t )
```

```
      ( all m: Node | m.prdc.(s.post) = m.prdc.t )
```

```
    } }
```

} frame
conditions:
nothing else
changes!

CHECKING THE INVARIANT

pred Invariant [t: Time] {

```
    OneOrderedRing [t]
    && ConnectedAppendages [t]
    && OrderedAppendages [t]
    && AntecedentPredecessors [t] }
```

Valid

further describe the reachable state space

assert StabilizationPreservesInvariant {

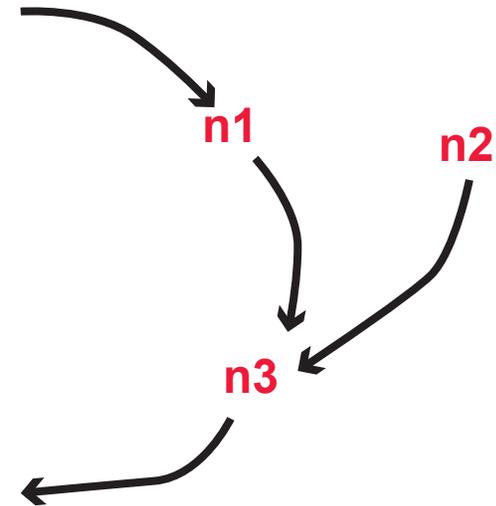
```
( Invariant [trace/first] ..... first event
  && some s: Stabilize, f: Notified | StabilizeCausesNotified [s, f]
) => Invariant [trace/last] }
```

check StabilizationPreservesInvariant for 5 but 2 Event, 3 Time

DEMONSTRATION

CHECKING ORDERED MERGES

```
pred OrderedMerges [t: Time] {  
  let ringMembers = {n: Node | n in n.(^(succ.t))} |  
  all disj n1, n2, n3: Node |  
    (  
      n1 in ringMembers && n3 in ringMembers  
      && n2 ! in ringMembers  
      && n3 in n1.succ.t && n3 in n2.succ.t  
    ) => Between[n1,n2,n3] }
```



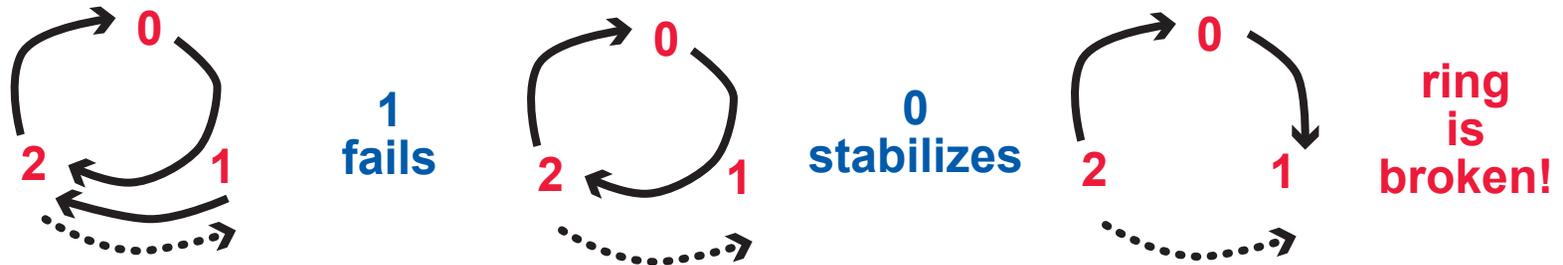
```
assert StabilizationPreservesOrderedMerges {  
  (  
    Invariant [trace/first]  
    && some s: Stabilize, f: Notified | StabilizeCausesNotified [s,f]  
  ) => OrderedMerges [trace/last] }
```

check StabilizationPreservesOrderedMerges for 3 but 2 Event, 3 Time

DEMONSTRATION

MAKING CHORD CORRECT, PART 1

HERE IS A SIMPLE CHORD BUG:

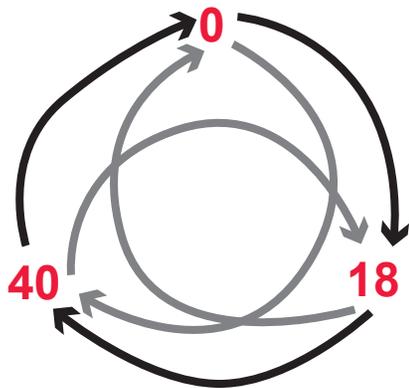


THERE ARE MANY SUCH BUGS IN THE ORIGINAL SPECIFICATION

FIX THEM BY BEING MORE DILIGENT ABOUT:

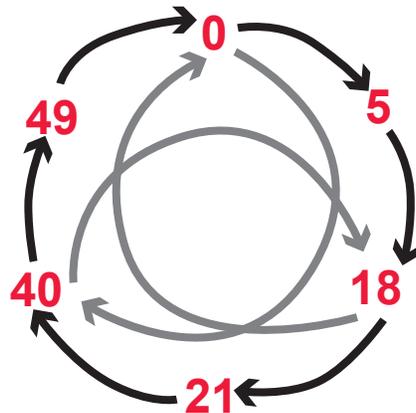
- checking that a node is live before replacing a good pointer with a pointer to it
- performing a reconcile (to get successor's successor list) whenever a node gets a new successor

ANOTHER CLASS OF COUNTEREXAMPLES

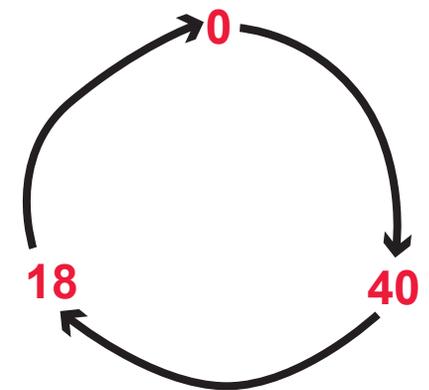


this network is ideal

3 nodes join and become integrated



new nodes fail, old nodes update



this network is disordered, and the protocol cannot fix it

this is a class of counterexamples:

- any ring of odd size becomes disordered
- any ring of even size splits into two disconnected subnetworks (which is another problem that the protocol cannot fix)

Chord has no specified timing constraints. This looks like a timing problem. Add timing constraints?



Maybe a good approach. I wasn't sure what timing constraints are enforceable. Can't constrain joins and failures.



For better or for worse, my version does not require timing constraints for correctness.

MAKING CHORD CORRECT, PART 2

MUST ANALYZE OPERATIONS IN TERMS OF ATOMIC EVENTS

operation at X
may change state of X

node X



node Y



an operation at X usually requires information from another node Y; X sends a query to Y



operation can be assumed to occur at this instant

if Y does not reply before a timeout at X, it is assumed that Y is dead or has left the network

node Z



the operation at X can also require information from another node Z



in this case the operation is two atomic events that can be interleaved with other events

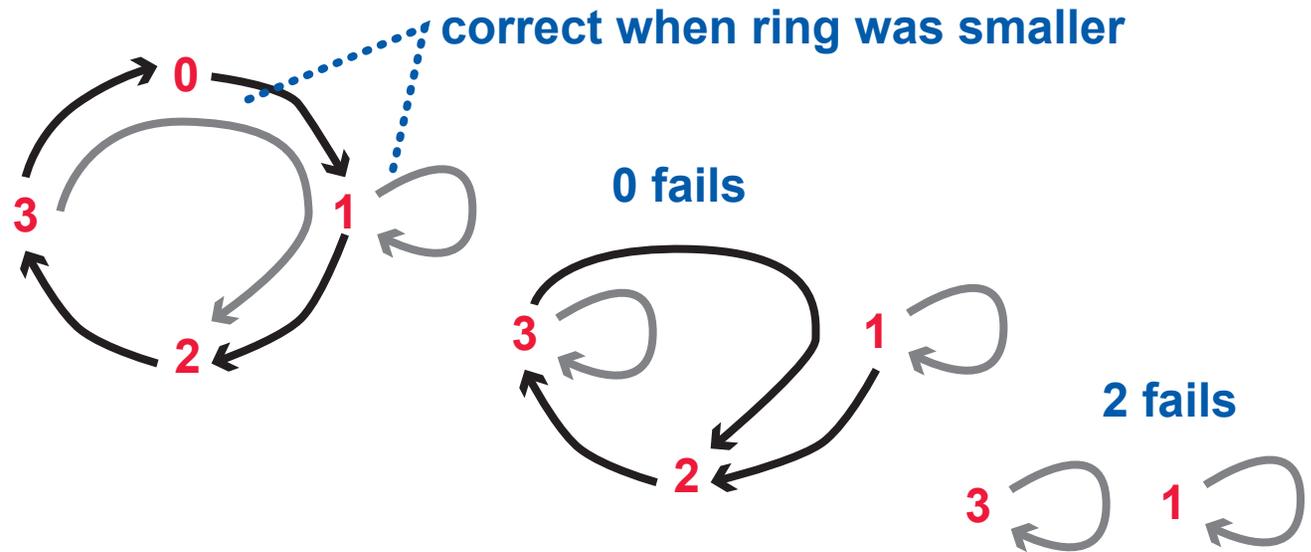
MAKING CHORD CORRECT, PART 3

THERE ARE STILL PROBLEMS WHEN ...

... a node fails or leaves, then rejoins when some node still has a pointer to it

the pointer is obsolete and wrong, but this cannot be detected because the node is live

... a node ends up pointing to itself



PROHIBIT NODE FROM REJOINING WITH ITS OLD NAME?

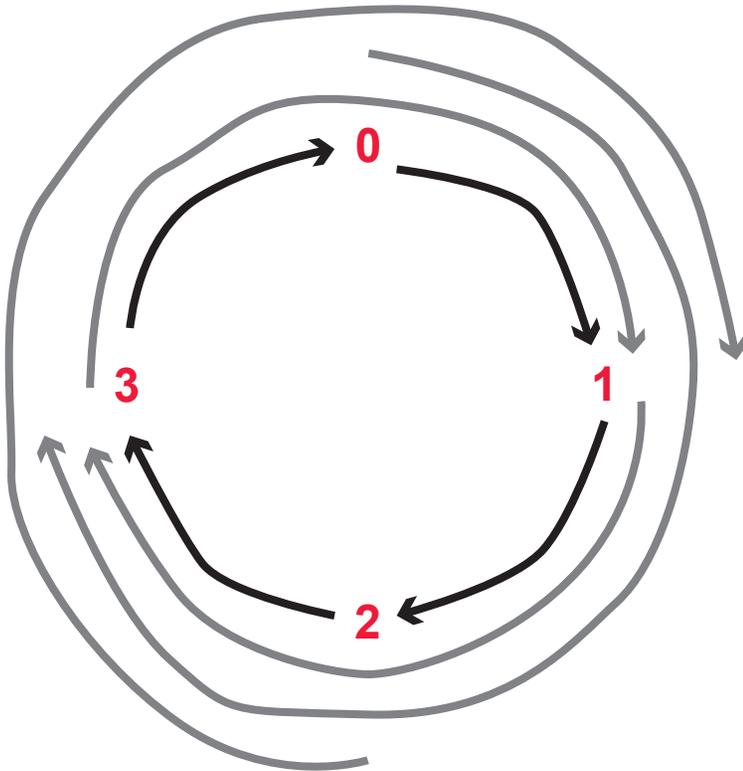
REQUIRE A MINIMUM RING SIZE OF SUCCESSOR-LIST-LENGTH + 1?

INDIVIDUALLY, NEITHER MAKES CHORD CORRECT

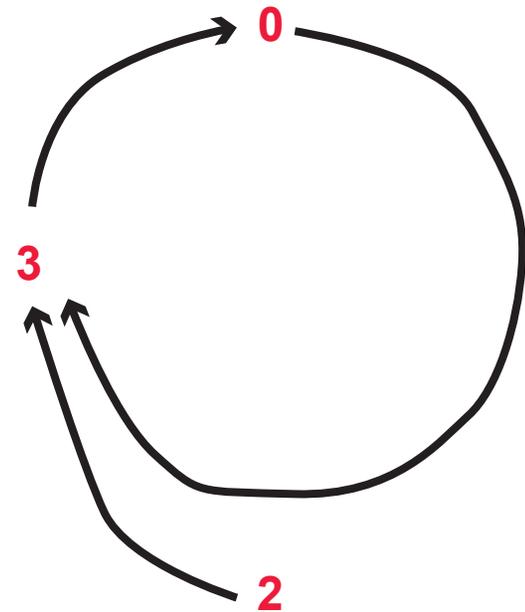
IT IS DIFFICULT TO MAINTAIN A MINIMUM RING SIZE

minimum ring size = 3

here ring size = 4



node 1
fails,
which
should
be acceptable



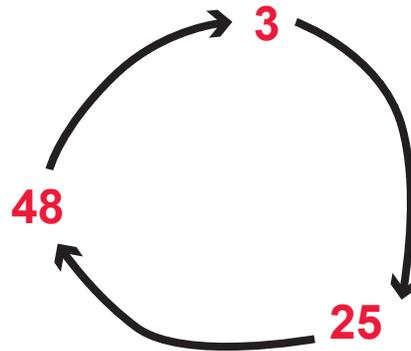
but actually,
the ring size
is now 2

MAKING CHORD CORRECT, PART 4

If a Chord network has
a *permanent* base of size . . .

successor-list-length + 1

. . . then it is provably correct.



network must be initialized
with these nodes

the machines at these IP
addresses (from which
the identifiers were
computed) should be
highly available . . .

. . . but even the
initialization helps a lot

*for example, need a base
of 5 to 10 machines—out of millions
in a peer-to-peer network*

NICE BENEFITS

- no timing constraints
- node can rejoin with an old identifier
- proof based on realistic assumptions about atomicity

PROOF OUTLINE

THEOREM: In any reachable state, if there are no subsequent joins or failures, then eventually the network will become ideal and remain ideal.

PROOF:

- 1** Define an invariant and show that it is true of all reachable states.
- 2** An operation that takes 0 or 1 query can be considered atomic. For operations that take 2 queries, show that the first half and the second half can safely be separated by other operations.
- 3** An effective repair operation is one that changes the network state. Define a natural-valued measure of the error in the network, and show that every effective repair operation decreases the error.
- 4** Show that whenever the network is not *ideal*, some effective repair operation is enabled.
- 5** Show that whenever the network is *ideal*, no effective repair operation is enabled.

not very demanding!

call it “eventual reachability”

PROVING THAT “INVARIANT” IS TRUE OF ALL REACHABLE STATES

```
assert InvariantInitiallyTrue {
```

```
  Initial[trace/first] => Invariant[trace/first]
}
```

check InvariantInitiallyTrue for 5 but 0 Event, 1 Time

```
assert JoinPreservesInvariant {
```

```
  some Join && Invariant[trace/first] => Invariant[trace/last]
}
```

check JoinPreservesInvariant for 5 but 1 Event, 2 Time

must repeat this for
the six other operations

includes all
nodes:
dead,
ring,
appendage

Alloy Analyzer says:

*No counterexamples found.
Assertion may be valid.*

What does that mean?

SMALL SCOPE HYPOTHESIS

NETWORK SIZE

We can only do exhaustive search for networks up to some size limit.

The “small scope hypothesis” makes explicit a folk theorem that most real bugs have small counterexamples.

Well-supported by experience, it is the philosophical basis of lightweight modeling and analysis.

RING STRUCTURES

The hypothesis is especially credible in this study, because ring structures are so symmetrical.

For example, to verify assertions relating pairs of nodes, it is only necessary to check rings of up to size 4 [Emerson & Namjoshi 95].

not directly relevant to Chord

EXPLORATION OF CHORD MODELS CONFIRMS THIS

Original version of Chord has minimum ring size of 1.

new counterexamples were found at network sizes 2, 3, 4 (many of each), and 5 (just one)

Correct version of Chord has minimum ring size of 3.

in exploring other versions with this minimum ring size, new counterexamples were found at network sizes 4, 5 (many of each), and 6 (just one)

WHAT SCOPE IS BIG ENOUGH?

The Alloy Analyzer can easily analyze networks up to size 8, and I stopped there.

PROOF OUTLINE

THEOREM: In any reachable state, if there are no subsequent joins or failures, then eventually the network will become ideal and remain ideal.

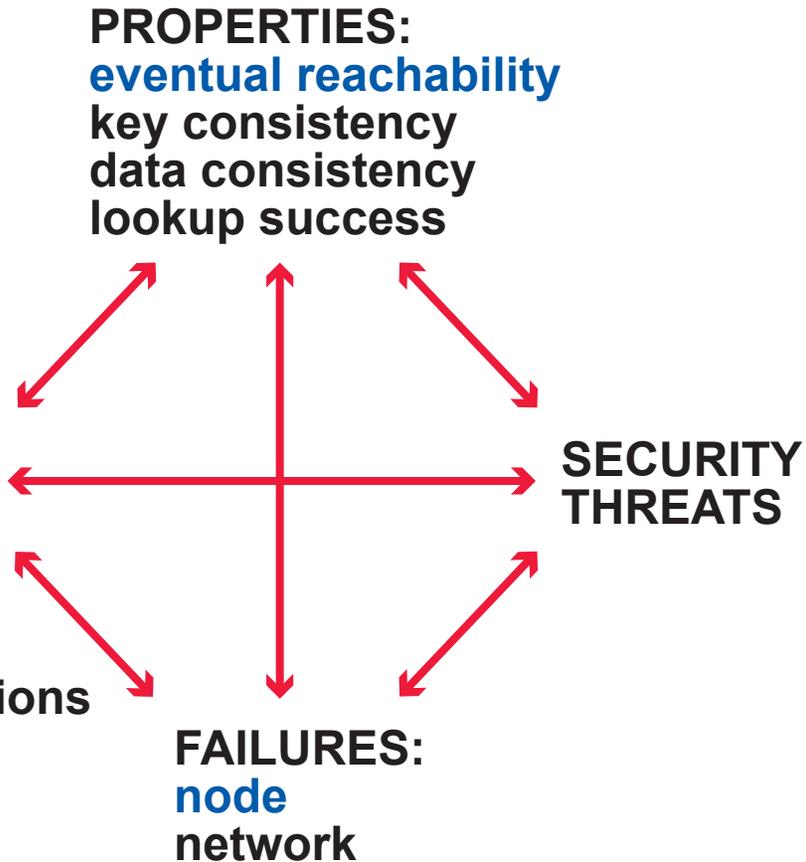
PROOF:

- 1** Define an invariant and show that it is true of all reachable states. **AUTOMATED** (exhaustive search over a finite domain)
- 2** An operation that takes 0 or 1 query can be considered atomic. For operations that take 2 queries, show that the first half **AUTOMATED** and the second half can safely be separated by other operations.
- 3** An effective repair operation is one **MANUAL** that changes the network state. Define a natural-valued measure of the error in the network, and show that every effective repair operation decreases the error.
- 4** Show that whenever the network is not *Ideal*, some effective repair operation is enabled. **AUTOMATED**
because the error is finite, after a finite number of repairs, the network will have no error and be Ideal
- 5** Show that whenever the network is *Ideal*, no effective repair operation is enabled. **AUTOMATED**
once it is ideal it stays ideal, because repair operations will not change it

FUTURE WORK

there are many other relationships to understand!

TECHNIQUES:
fuller population
fresh identifiers
minimum size
stable base
data replication
timing constraints
probability distributions
(good luck!)



for subtle protocols like this,
formal modeling and
automated analysis may not
be sufficient, but they are . . .

. . . ABSOLUTELY NECESSARY

REFERENCES

ANYTHING YOU WANT TO KNOW ABOUT ALLOY

alloy.mit.edu

CHORD CORRECTNESS

“Using lightweight modeling to understand Chord,” Pamela Zave,
ACM SIGCOMM Computer Communications Review, April 2012.

“A practical comparison of Alloy and Spin,” Pamela Zave,
submitted for publication.

www2.research.att.com/~pamela/chord.html