CPSC 416 - Distributed program analysis and Invariant Inference

March 28, 2018

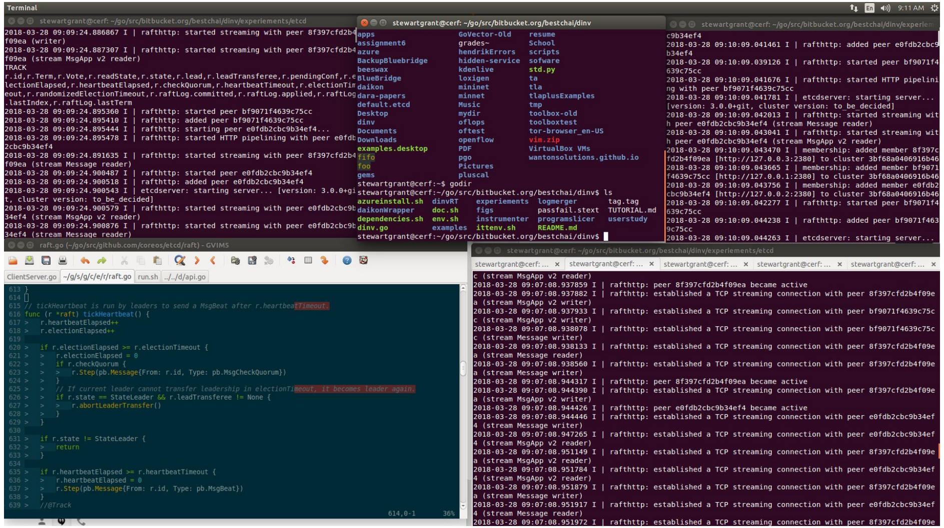
Lecturer Stewart Grant

The problem: distributed systems are complex!

Difficulty	**Factors**
Understandability	● Concurrency ● Decentralized State ● No centralized clock
Debug	● Nondeterminism ● Changing environment (network)
Test	● State space is massive ● Exhaustive testing is impractical ● Configuration space is even larger ● Cost of large scale deployments

How do you know that a distributed system works?

- Logging
    - Open log in emacs/vi, brew coffee, get comfortable!
    - Maybe use ShiViz on the logs if you are debugging protocol issues
- Test as much as you can (Unit/Integration/**Stress**)
- Mathematically prove correct?
    - (No one does that really)



```
2018-03-28 09:09:24.886867 I | rafthttp: started streaming with peer 8f397cfd2b4
f09ea (writer)
2018-03-28 09:09:24.887307 I | rafthttp: started streaming with peer 8f397cfd2b4
f09ea (stream MsgApp v2 reader)
TRACK
r.id,r.Term,r.Vote,r.readState,r.state,r.lead,r.leadTransferee,r.pendingConf,r.e
lectionElapsed,r.heartbeatElapsed,r.checkQuorum,r.heartbeatTimeout,r.electionTim
eout,r.randomizedElectionTimeout,r.raftLog.committed,r.raftLog.applied,r.raftLog
.lastIndex,r.raftLog.lastTerm
2018-03-28 09:09:24.895360 I | rafthttp: started peer bf9071f4639c75cc
2018-03-28 09:09:24.895410 I | rafthttp: added peer bf9071f4639c75cc
2018-03-28 09:09:24.895444 I | rafthttp: starting peer e0fdb2cbc9b34ef4...
2018-03-28 09:09:24.895478 I | rafthttp: started HTTP pipelining with peer e0fdb
2cbc9b34ef4
2018-03-28 09:09:24.891635 I | rafthttp: started streaming with peer 8f397cfd2b4
f09ea (stream Message reader)
2018-03-28 09:09:24.900487 I | rafthttp: started peer e0fdb2cbc9b34ef4
2018-03-28 09:09:24.900518 I | rafthttp: added peer e0fdb2cbc9b34ef4
2018-03-28 09:09:24.900543 I | etcdserver: starting server... [version: 3.0.0+gi
t, cluster version: to_be_decided]
2018-03-28 09:09:24.900579 I | rafthttp: started streaming with peer e0fdb2cbc9b
34ef4 (stream MsgApp v2 reader)
2018-03-28 09:09:24.900876 I | rafthttp: started streaming with peer e0fdb2cbc9b
34ef4 (stream Message reader)
```

```
apps                   GoVector-Old    resume
assignment6            grades~         School
azure                  hendrikErrors   scripts
BackupBluebridge       hidden-service  sofware
beeswax                kdenlive        std.py
BlueBridge             loxigen         ta
daikon                 mininet         tla
dara-papers            minnet          tlaplusExamples
default.etcd           Music           tmp
Desktop                mydir           toolbox-old
dinv                   oflops          toolboxtest
Documents              oftest          tor-browser_en-US
Downloads              openflow        vim.zip
examples.desktop       PDF             VirtualBox VMs
fifo                   pgo             wantonsolutions.github.io
foo                    Pictures
gems                   pluscal
stewartgrant@cerf:~$ godir
stewartgrant@cerf:~/go/src/bitbucket.org/bestchai/dinv$ ls
azureinstall.sh  dinvRT    experiements  logmerger       tag.tag
daikonWrapper    doc.sh    figs          passfail.stext  TUTORIAL.md
dependencies.sh  env.sh    instrumenter  programslicer   userstudy
dinv.go          examples  ittenv.sh     README.md
stewartgrant@cerf:~/go/src/bitbucket.org/bestchai/dinv$
```

```
c9b34ef4
2018-03-28 09:10:09.041461 I | rafthttp: added peer e0fdb2cbc9
b34ef4
2018-03-28 09:10:09.039126 I | rafthttp: started peer bf9071f4
639c75cc
2018-03-28 09:10:09.041676 I | rafthttp: started HTTP pipelini
ng with peer bf9071f4639c75cc
2018-03-28 09:10:09.041781 I | etcdserver: starting server...
[version: 3.0.0+git, cluster version: to_be_decided]
2018-03-28 09:10:09.042013 I | rafthttp: started streaming wit
h peer e0fdb2cbc9b34ef4 (stream Message reader)
2018-03-28 09:10:09.043041 I | rafthttp: started streaming wit
h peer e0fdb2cbc9b34ef4 (stream MsgApp v2 reader)
2018-03-28 09:10:09.043470 I | membership: added member 8f397c
fd2b4f09ea [http://127.0.0.3:2380] to cluster 3bf68a0406916b46
2018-03-28 09:10:09.043665 I | membership: added member bf9071
f4639c75cc [http://127.0.0.1:2380] to cluster 3bf68a0406916b46
2018-03-28 09:10:09.043756 I | membership: added member e0fdb2
cbc9b34ef4 [http://127.0.0.2:2380] to cluster 3bf68a0406916b46
2018-03-28 09:10:09.042277 I | rafthttp: started peer bf9071f4
639c75cc
2018-03-28 09:10:09.044238 I | rafthttp: added peer bf9071f463
9c75cc
2018-03-28 09:10:09.044263 I | etcdserver: starting server...
```

```
}

// tickHeartbeat is run by leaders to send a MsgBeat after r.heartbeatTimeout.
func (r *raft) tickHeartbeat() {
>   r.heartbeatElapsed++
>   r.electionElapsed++

>   if r.electionElapsed >= r.electionTimeout {
>   >   r.electionElapsed = 0
>   >   if r.checkQuorum {
>   >   >   r.Step(pb.Message{From: r.id, Type: pb.MsgCheckQuorum})
>   >   }
>   >   // If current leader cannot transfer leadership in electionTimeout, it becomes leader again.
>   >   if r.state == StateLeader && r.leadTransferee != None {
>   >   >   r.abortLeaderTransfer()
>   >   }
>   }

>   if r.state != StateLeader {
>   >   return
>   }

>   if r.heartbeatElapsed >= r.heartbeatTimeout {
>   >   r.heartbeatElapsed = 0
>   >   r.Step(pb.Message{From: r.id, Type: pb.MsgBeat})
>   }
>   //@Track
                                                              614,0-1       36%
```

```
c (stream MsgApp v2 reader)
2018-03-28 09:07:08.937859 I | rafthttp: peer 8f397cfd2b4f09ea became active
2018-03-28 09:07:08.937882 I | rafthttp: established a TCP streaming connection with peer 8f397cfd2b4f09e
a (stream MsgApp v2 writer)
2018-03-28 09:07:08.937933 I | rafthttp: established a TCP streaming connection with peer bf9071f4639c75c
c (stream MsgApp v2 writer)
2018-03-28 09:07:08.938078 I | rafthttp: established a TCP streaming connection with peer bf9071f4639c75c
c (stream Message writer)
2018-03-28 09:07:08.938133 I | rafthttp: established a TCP streaming connection with peer 8f397cfd2b4f09e
a (stream Message reader)
2018-03-28 09:07:08.938560 I | rafthttp: established a TCP streaming connection with peer 8f397cfd2b4f09e
a (stream Message writer)
2018-03-28 09:07:08.944317 I | rafthttp: peer 8f397cfd2b4f09ea became active
2018-03-28 09:07:08.944390 I | rafthttp: established a TCP streaming connection with peer 8f397cfd2b4f09e
a (stream MsgApp v2 writer)
2018-03-28 09:07:08.944426 I | rafthttp: peer e0fdb2cbc9b34ef4 became active
2018-03-28 09:07:08.944446 I | rafthttp: established a TCP streaming connection with peer e0fdb2cbc9b34ef
4 (stream Message writer)
2018-03-28 09:07:08.947265 I | rafthttp: established a TCP streaming connection with peer e0fdb2cbc9b34ef
4 (stream MsgApp v2 reader)
2018-03-28 09:07:08.951149 I | rafthttp: established a TCP streaming connection with peer 8f397cfd2b4f09e
a (stream MsgApp v2 reader)
2018-03-28 09:07:08.951784 I | rafthttp: established a TCP streaming connection with peer e0fdb2cbc9b34ef
4 (stream MsgApp v2 reader)
2018-03-28 09:07:08.951879 I | rafthttp: established a TCP streaming connection with peer 8f397cfd2b4f09e
a (stream Message writer)
2018-03-28 09:07:08.951917 I | rafthttp: established a TCP streaming connection with peer e0fdb2cbc9b34ef
4 (stream Message reader)
2018-03-28 09:07:08.951972 I | rafthttp: established a TCP streaming connection with peer 8f397cfd2b4f09e
```

Figure (1) A typical distributed systems developers desktop [my desktop]

What other techniques are available?

Static analysis:

Analyze a programs source code, without running the program.

- Type Checker
- Linter
- Symbolic Execution

Complete but over-approximate and expensive.

Dynamic analysis:

Analyze a programs behavior as it runs, usually by logging.

- Testing
- Profiling
- Deadlock detection
- Memory profiling (valgrind)

Incomplete but scalable

Today's lecture

- Program analysis background
    - Static analysis
    - Dynamic analysis
- Dinv's tool and analyses
    - Data invariants
    - Static: program slicing
    - Dynamic: distributed lattice construction
- Answer any Dinv questions you might have

Program Properties: Data invariants

- An invariant is a property that holds on data at all times
- A data invariant can hold between 1 or more variables
- Data invariants are type dependent

Knowledge of a programs invariants is important for understanding if it is correct or faulty.

Example Program:

```
var sum = 0
for i:=0;i<TOTAL;i++{
     sum += i
}
```

Example Invariants:

```
i < TOTAL // loop invariant
i >= 0
sum >= i
```

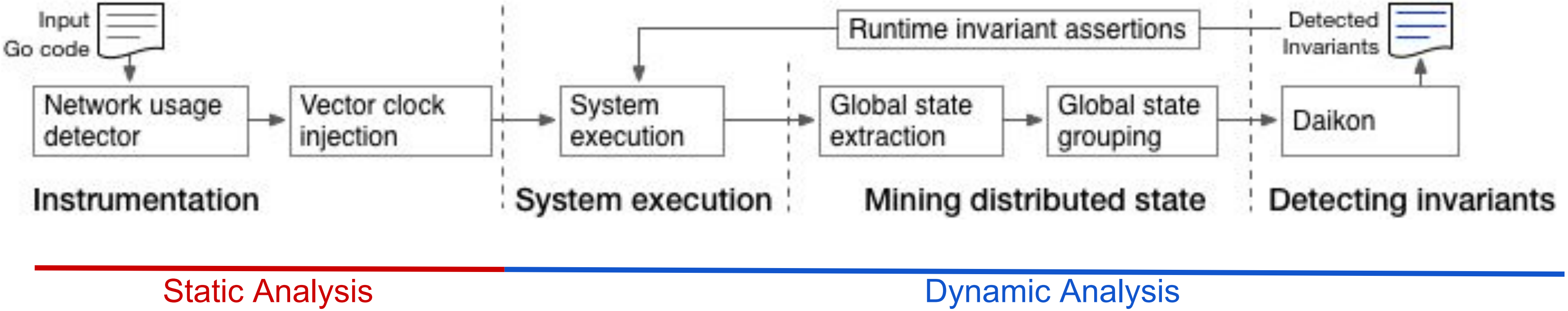
What is a Distributed Data Invariant?

- Distributed data invariants hold across 1 or more nodes in a distributed system
- Some hold globally at all times
- Some are protocol specific

Ex) Distributed Key Value Store Invariant. No two nodes serve the same keys.

$$\forall Nodes\ i, j,\ Keys_i \neq Keys_j$$

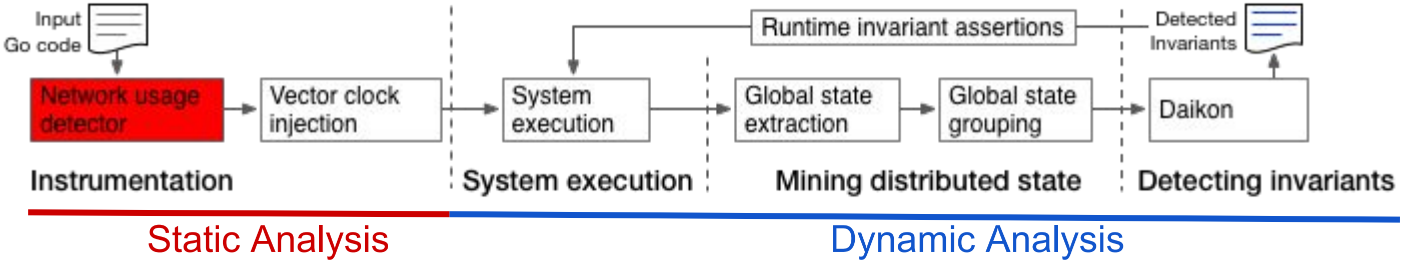
Dinv Overview



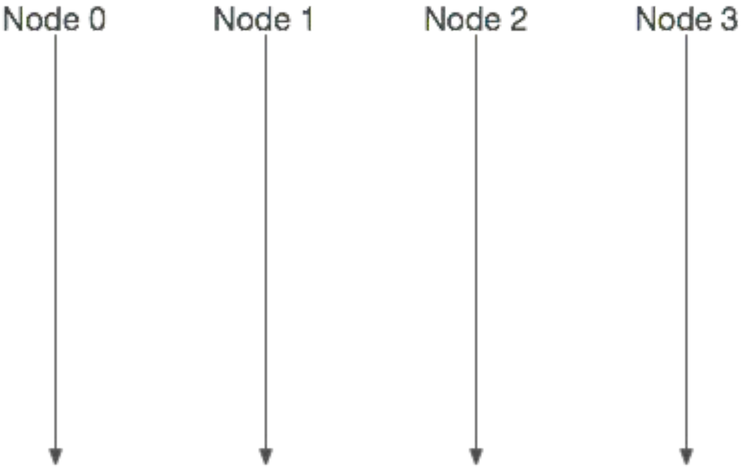
1. Distributed Invariant Inference Challenges
   - a. **What state should be logged and when?**
   - b. How to infer distributed invariants from logged state?
   - c. How to enforce distributed invariants?

What variables should be logged?

- **Massive** variable state space
- Exponentially larger invariant state space
- Arbitrary distributed invariants be minimized

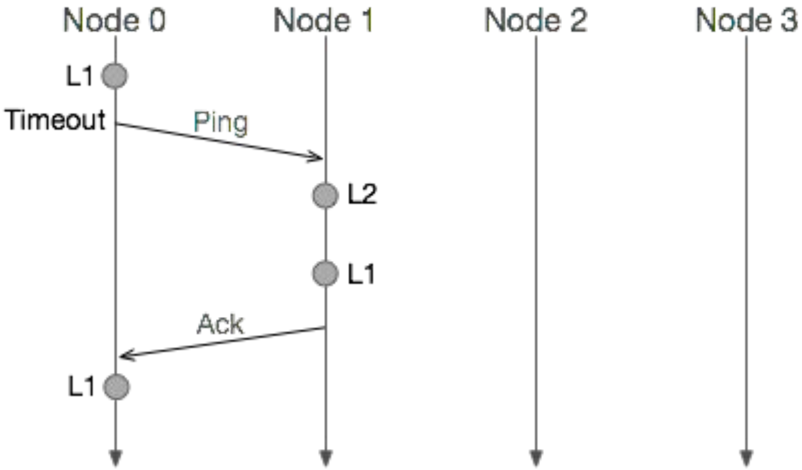


Example Code: Serf



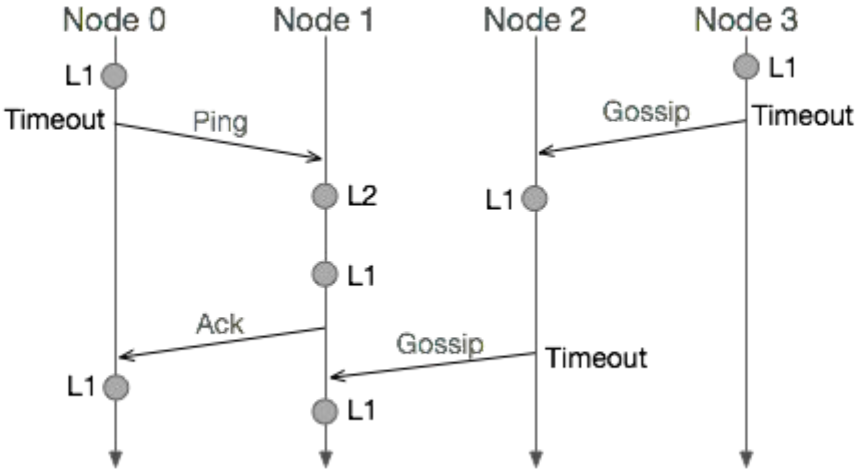
```
func (s serfNode) serf(conn UDPConnection) {
 for true {
















}}}
```

Example Code: Serf



```
func (s serfNode) serf(conn UDPConnection) {
 for true {
   msg := conn.Read()
   switch msg.Type {
   case PING:
     conn.WriteToUDP("ACK", msg.Sender)
     break


   }
   timeout := s.CheckForTimeouts()
   switch timeout.Type {
   case PING:
     conn.WriteToUDP("PING",timeout.Node)
     break



 }}}
```

Example Code: Serf

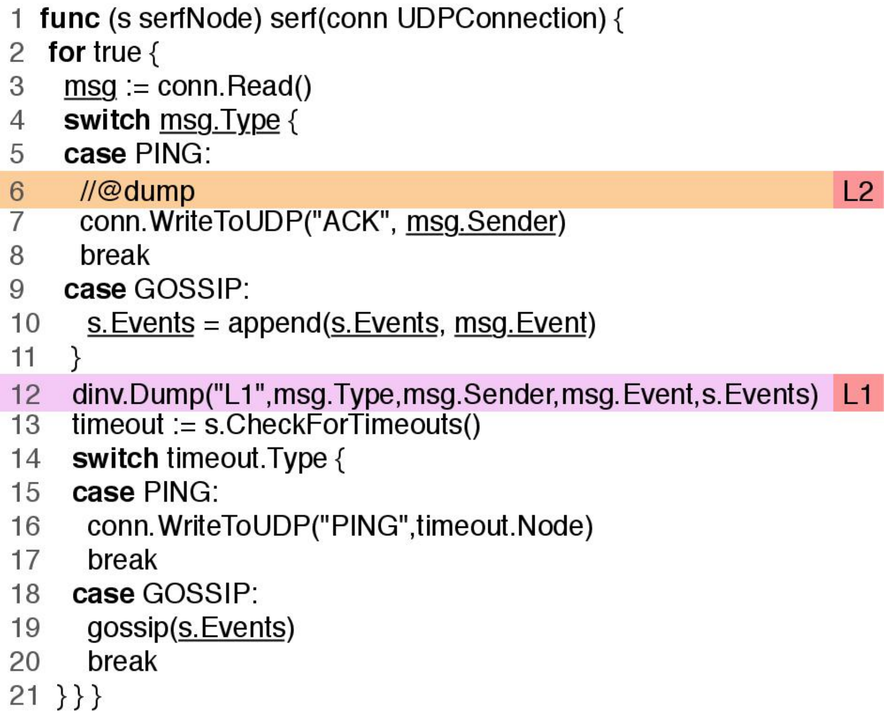


```
func (s serfNode) serf(conn UDPConnection) {
 for true {
  msg := conn.Read()
  switch msg.Type {
  case PING:
   conn.WriteToUDP("ACK", msg.Sender)
   break
  case GOSSIP:
   s.Events = append(s.Events, msg.Event)
  }
  timeout := s.CheckForTimeouts()
  switch timeout.Type {
  case PING:
   conn.WriteToUDP("PING",timeout.Node)
   break
  case GOSSIP:
   gossip(s.Events)
   break
}}}
```

What state should be logged and when?

Insight: *Important distributed state must have dataflow to and from the network*.

Technique: Program slicing [Ottenstein 84]



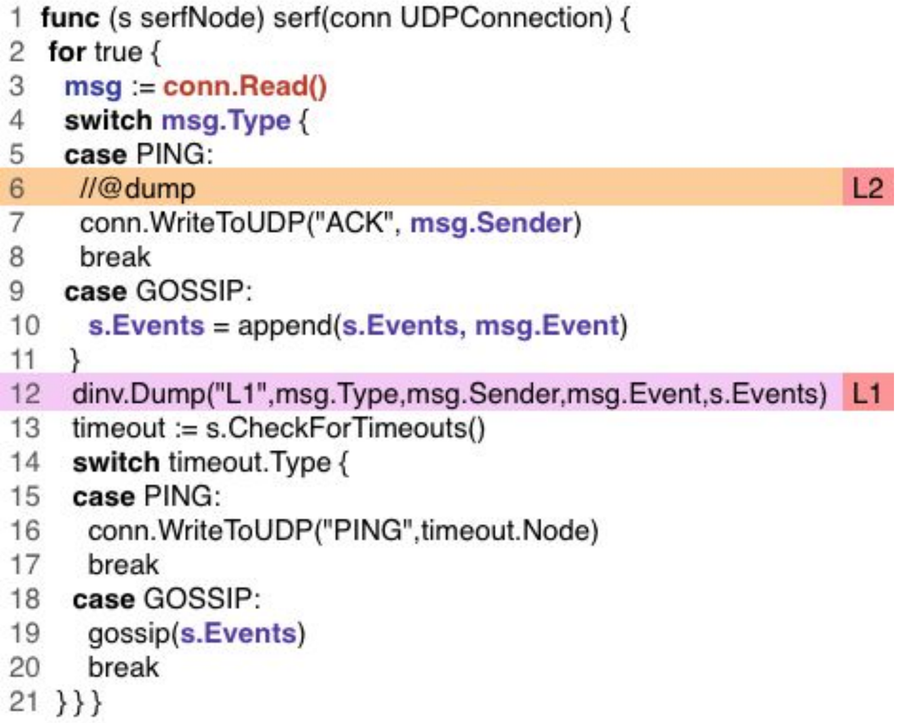
```
func (s serfNode) serf(conn UDPConnection) {
 for true {
  msg := conn.Read()
  switch msg.Type {
  case PING:
    //@dump                                                    L2
    conn.WriteToUDP("ACK", msg.Sender)
    break
  case GOSSIP:
    s.Events = append(s.Events, msg.Event)
  }
  dinv.Dump("L1",msg.Type,msg.Sender,msg.Event,s.Events)   L1
  timeout := s.CheckForTimeouts()
  switch timeout.Type {
  case PING:
    conn.WriteToUDP("PING",timeout.Node)
    break
  case GOSSIP:
    gossip(s.Events)
    break
}}}
```

What state should be logged and when?

Insight: *Important distributed state must have dataflow to and from the network*.

Technique: Program slicing [Ottenstein 84]

- Transitively track assignments to variables
- A slice is the complete set of statements over which marked data flows



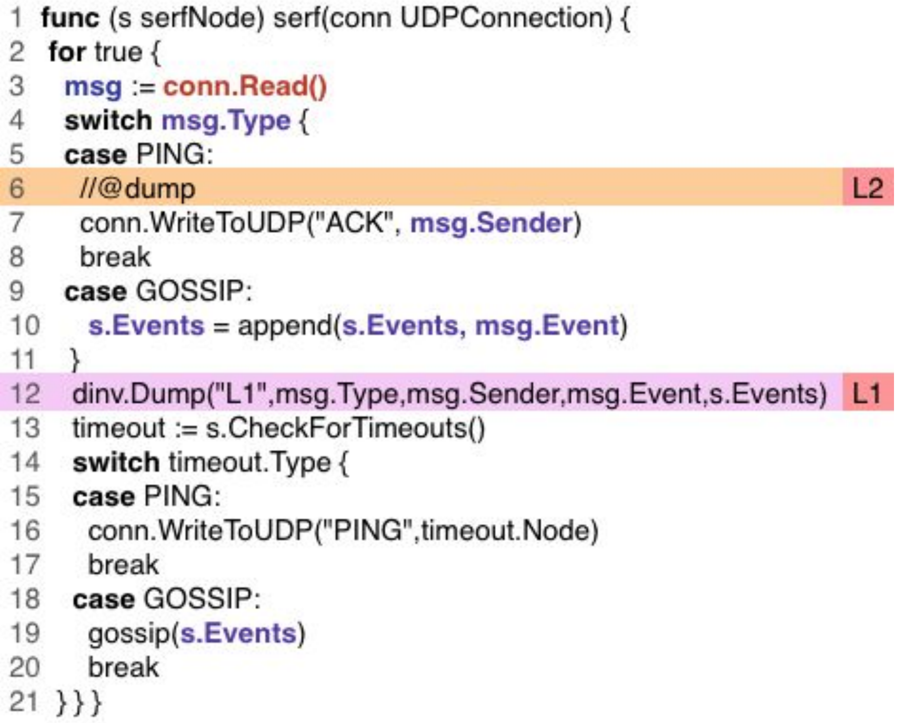
```
func (s serfNode) serf(conn UDPConnection) {
 for true {
  msg := conn.Read()
  switch msg.Type {
  case PING:
   //@dump                                              L2
   conn.WriteToUDP("ACK", msg.Sender)
   break
  case GOSSIP:
   s.Events = append(s.Events, msg.Event)
  }
  dinv.Dump("L1",msg.Type,msg.Sender,msg.Event,s.Events) L1
  timeout := s.CheckForTimeouts()
  switch timeout.Type {
  case PING:
   conn.WriteToUDP("PING",timeout.Node)
   break
  case GOSSIP:
   gossip(s.Events)
   break
}}}
```

What state should be logged and when?

Insight: *Important distributed state must have dataflow to and from the network*.

Technique: Program slicing [Ottenstein 84]

- Transitively track assignments to variables
- A slice is the complete set of statements over which marked data flows



```
func (s serfNode) serf(conn UDPConnection) {
 for true {
   msg := conn.Read()
   switch msg.Type {
   case PING:
     //@dump
     conn.WriteToUDP("ACK", msg.Sender)
     break
   case GOSSIP:
     s.Events = append(s.Events, msg.Event)
   }
   dinv.Dump("L1",msg.Type,msg.Sender,msg.Event,s.Events)
   timeout := s.CheckForTimeouts()
   switch timeout.Type {
   case PING:
     conn.WriteToUDP("PING",timeout.Node)
     break
   case GOSSIP:
     gossip(s.Events)
     break
}}}
```

Q: What do you think the challenges of dataflow analysis are?

Some Answers:

- Aliasing (when one bit of data can be confused with many)
- Pointer analysis
- Interprocedural flow
- Thread interleaving
- Distributed dataflow

Where should state be logged?

Location and frequency of logging correspond to invariant accuracy

Instrumentation Strategy	**Location choice**	**Variable Choice**
Function entrances/exits	Auto	Auto
Network calls	Auto	Auto
User-defined annotations	Manual	Manual or auto

How to Instrument with Dinv

Source: $REPOLOCATION/dinv/examples/helloDinv/ClientServer.go

Pre Instrumentation:

Two annotations:

@Track & @Dump

Track Recommended (Reduces Output Size)

```
func client(listen, send string) {
>   // sending UDP packet to specified address and port
>   conn := setupConnection(SERVERPORT, CLIENTPORT)
>   for i := 0; i < MESSAGES; i++ {
>   >   //@track
>   >   outgoingMessage := i
>   >   outBuf := dinvRT.Pack(outgoingMessage)
>   >   _, errWrite := conn.Write(outBuf)
>   >   printErr(errWrite)
>   >   var inBuf [512]byte
>   >   var incommingMessage int
>   >   n, errRead := conn.Read(inBuf[0:])
>   >   printErr(errRead)
>   >   dinvRT.Unpack(inBuf[0:n], &incommingMessage)
>   >   incommingMessage = n - n + incommingMessage
>   >   //fmt.Printf("GOT BACK : %d\n", incommingMessage)
>   >   time.Sleep(1)
>   }
>   done <- 1
}
```

How to Instrument with Dinv

Source: $REPOLOCATION/dinv/examples/helloDinv/ClientServer.go

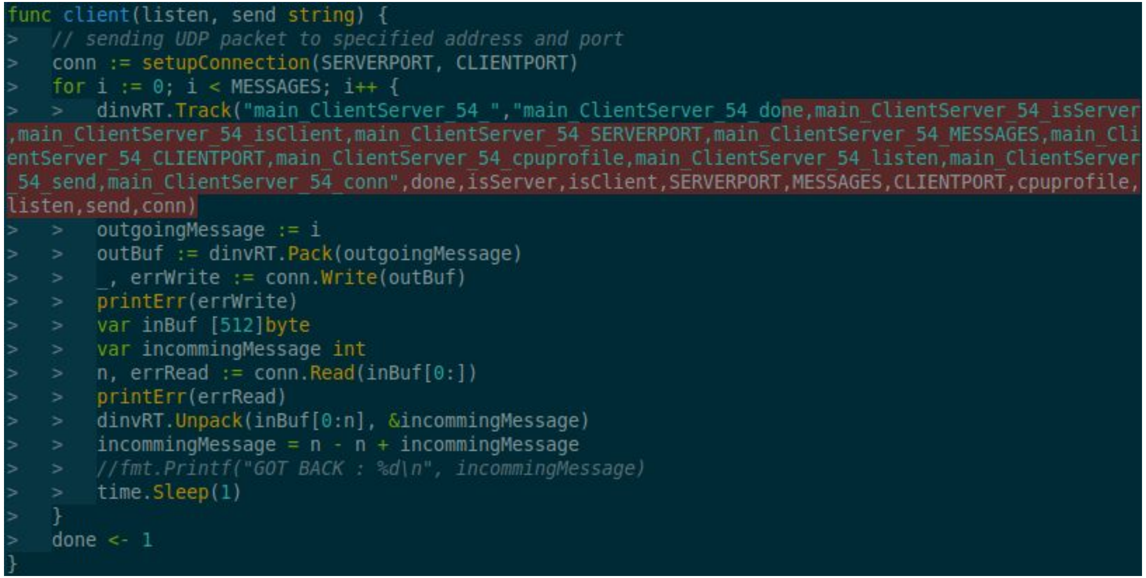
Instrumentation Command:

```
dinv -i
-file=ClientServer.go
```

The resulting source code is Instrumented to log variables.

Revert Instrumentation

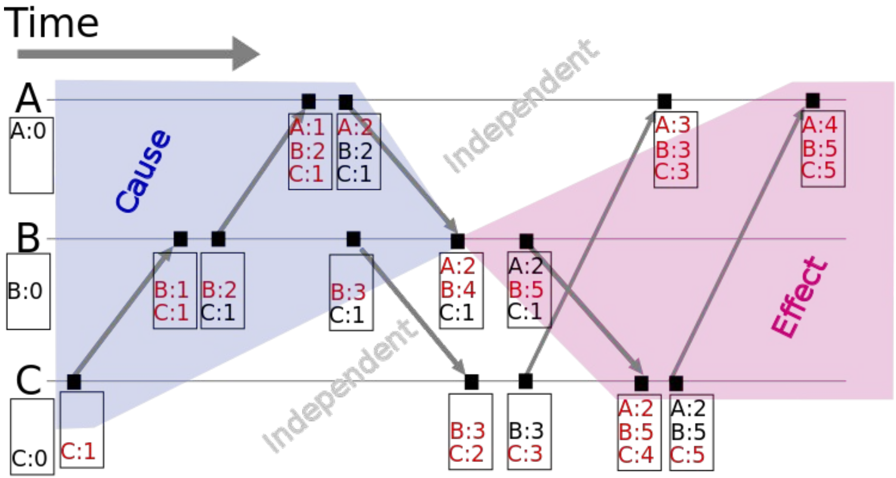
```
dinv -i -c
-file=ClientServer.go
```



```
func client(listen, send string) {
>   // sending UDP packet to specified address and port
>   conn := setupConnection(SERVERPORT, CLIENTPORT)
>   for i := 0; i < MESSAGES; i++ {
>   >   dinvRT.Track("main_ClientServer_54_","main_ClientServer_54_done,main_ClientServer_54_isServer
,main_ClientServer_54_isClient,main_ClientServer_54_SERVERPORT,main_ClientServer_54_MESSAGES,main_Cli
entServer_54_CLIENTPORT,main_ClientServer_54_cpuprofile,main_ClientServer_54_listen,main_ClientServer
_54_send,main_ClientServer_54_conn",done,isServer,isClient,SERVERPORT,MESSAGES,CLIENTPORT,cpuprofile,
listen,send,conn)
>   >   outgoingMessage := i
>   >   outBuf := dinvRT.Pack(outgoingMessage)
>   >   _, errWrite := conn.Write(outBuf)
>   >   printErr(errWrite)
>   >   var inBuf [512]byte
>   >   var incommingMessage int
>   >   n, errRead := conn.Read(inBuf[0:])
>   >   printErr(errRead)
>   >   dinvRT.Unpack(inBuf[0:n], &incommingMessage)
>   >   incommingMessage = n - n + incommingMessage
>   >   //fmt.Printf("GOT BACK : %d\n", incommingMessage)
>   >   time.Sleep(1)
>   }
>   done <- 1
}
```

Vector clock refresher

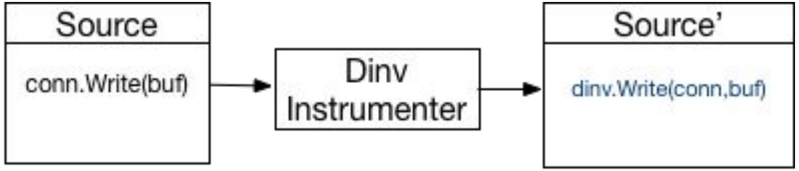
- Distributed systems lack a centralized clock
- Ordering events is therefore hard
- Fundamentally the best that can be done is a *partial order* with happens before
- If **A** receives a message from **B**, the sending event on **B** *happened before* the receive event on **A**



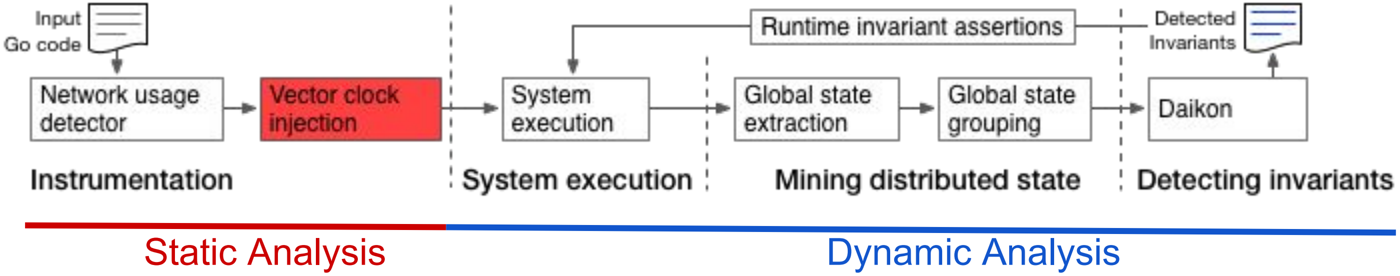
Algorithm:
1) Increment own index on send & receive
2) Take max of all indexes on receive

Tracking time: Vector Clock Instrumentation

- Establish partial event ordering
- Manual and automatic options
- Covers Go standard *net* library



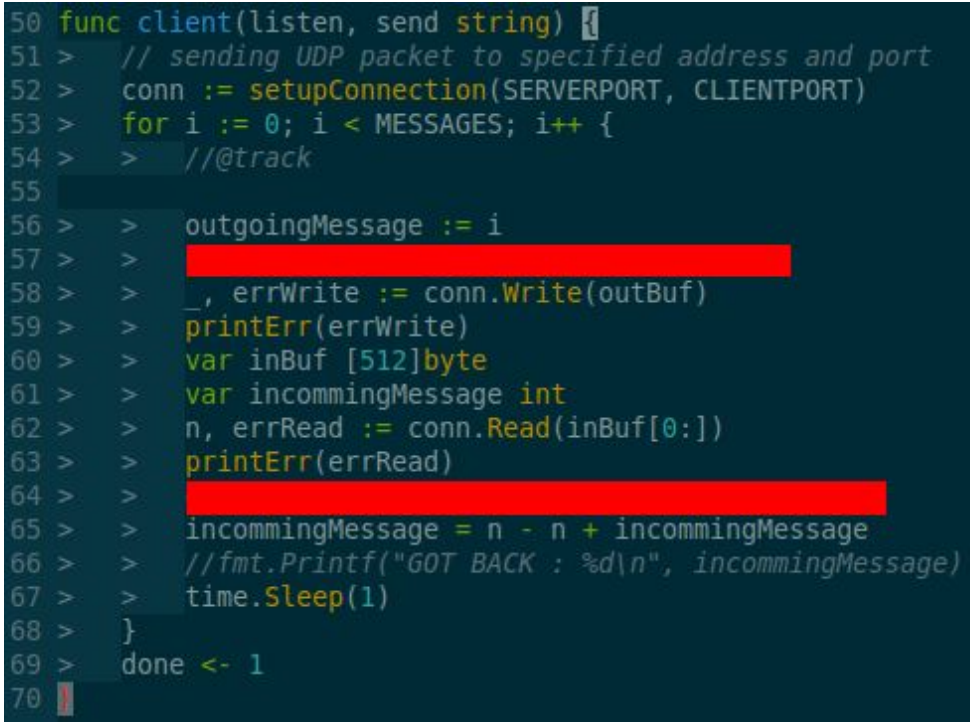
Repository: https://github.com/DistributedClocks



Example Vector Clock: Pack/Unpack

Pre-manual Instrumentation:

- network write line 58
- Network read line 62



```
50 func client(listen, send string) {
51 >   // sending UDP packet to specified address and port
52 >   conn := setupConnection(SERVERPORT, CLIENTPORT)
53 >   for i := 0; i < MESSAGES; i++ {
54 >   >   //@track
55
56 >   >   outgoingMessage := i
57 >   >   [redacted]
58 >   >   _, errWrite := conn.Write(outBuf)
59 >   >   printErr(errWrite)
60 >   >   var inBuf [512]byte
61 >   >   var incommingMessage int
62 >   >   n, errRead := conn.Read(inBuf[0:])
63 >   >   printErr(errRead)
64 >   >   [redacted]
65 >   >   incommingMessage = n - n + incommingMessage
66 >   >   //fmt.Printf("GOT BACK : %d\n", incommingMessage)
67 >   >   time.Sleep(1)
68 >   }
69 >   done <- 1
70 }
```

Example Vector Clock: Pack/Unpack

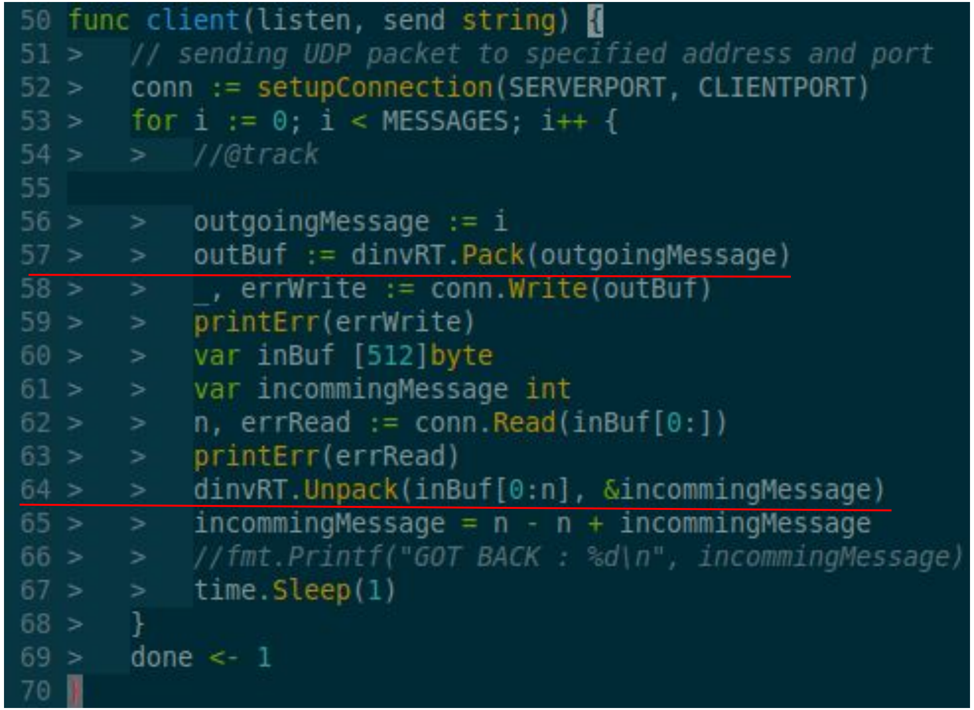
Pre-manual Instrumentation:

- Pack line 57

`Pack(payload interface{}) []byte`

- Unpack line 64

`Unpack(buf []byte, toFill interface{})`



```
50 func client(listen, send string) {
51 >   // sending UDP packet to specified address and port
52 >   conn := setupConnection(SERVERPORT, CLIENTPORT)
53 >   for i := 0; i < MESSAGES; i++ {
54 >   >   //@track
55
56 >   >   outgoingMessage := i
57 >   >   outBuf := dinvRT.Pack(outgoingMessage)
58 >   >   _, errWrite := conn.Write(outBuf)
59 >   >   printErr(errWrite)
60 >   >   var inBuf [512]byte
61 >   >   var incommingMessage int
62 >   >   n, errRead := conn.Read(inBuf[0:])
63 >   >   printErr(errRead)
64 >   >   dinvRT.Unpack(inBuf[0:n], &incommingMessage)
65 >   >   incommingMessage = n - n + incommingMessage
66 >   >   //fmt.Printf("GOT BACK : %d\n", incommingMessage)
67 >   >   time.Sleep(1)
68 >   }
69 >   done <- 1
70 }
```

VC Instrumentation Options

- Dinv Pack/Unpack take care of marshalling structs!
  - Allows for custom messages to be logged along side vector clocks
- Govector automatically instruments if marshalling is already done
  - Automatic!

GoVector Repository
https://github.com/DistributedClocks/GoVector

Command:

```
GoVector -file=filename
```

Method of Injection: AST Rotation

Before:

```
Err = conn.Write(buf)
```

After

```
GoVector.Write(conn.Write,buf)
```

Example Output

GoVector filename format <nodename>.log-Log.txt

Example Govector output:

```
821589986 {"821589986":1}
Initialization Complete
821589986 {"821589986":2}
Sending from 821589986_main.client+0xaa_/home/stewartgrant/go/src/bitbucket.org/bestchai/dinv/examples/helloDinv/ClientServer.go:57 821589986
821589986 {"821589986":3, "822468001":3}
Received on 821589986_main.client+0x254_/home/stewartgrant/go/src/bitbucket.org/bestchai/dinv/examples/helloDinv/ClientServer.go:64 821589986
821589986 {"822468001":3, "821589986":4}
Sending from 821589986_main.client+0xaa_/home/stewartgrant/go/src/bitbucket.org/bestchai/dinv/examples/helloDinv/ClientServer.go:57 821589986
821589986 {"822468001":5, "821589986":5}
Received on 821589986_main.client+0x254_/home/stewartgrant/go/src/bitbucket.org/bestchai/dinv/examples/helloDinv/ClientServer.go:64 821589986
```

ShiViz.sh and dinv-shiviz

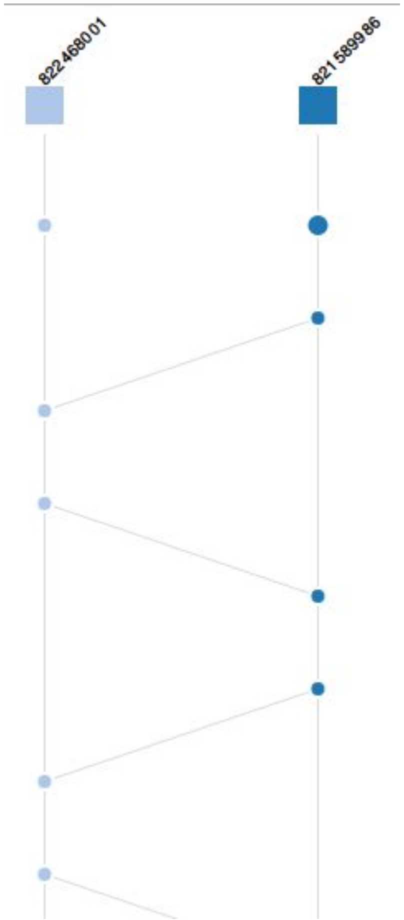
Communication patterns can be visualized by merging clock files.

To generate ShiViz parseable logs run either

```
$PATHTOREPO/dinv/ShiViz.sh
```

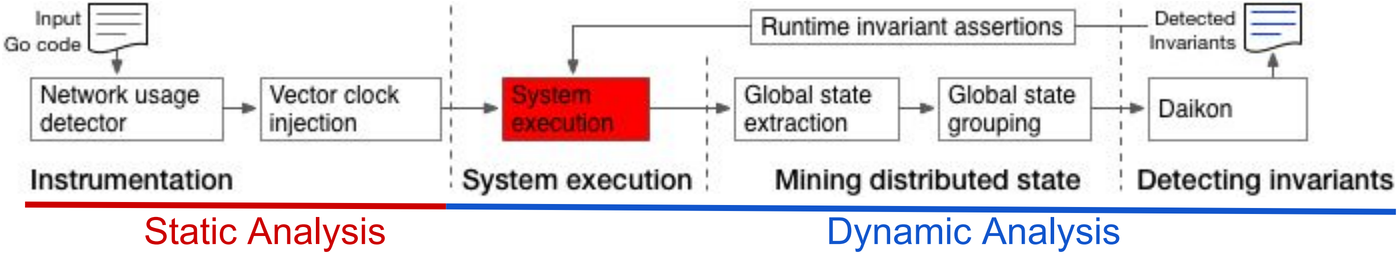
Or

```
dinv -l -shiviz *log-Log.txt
```



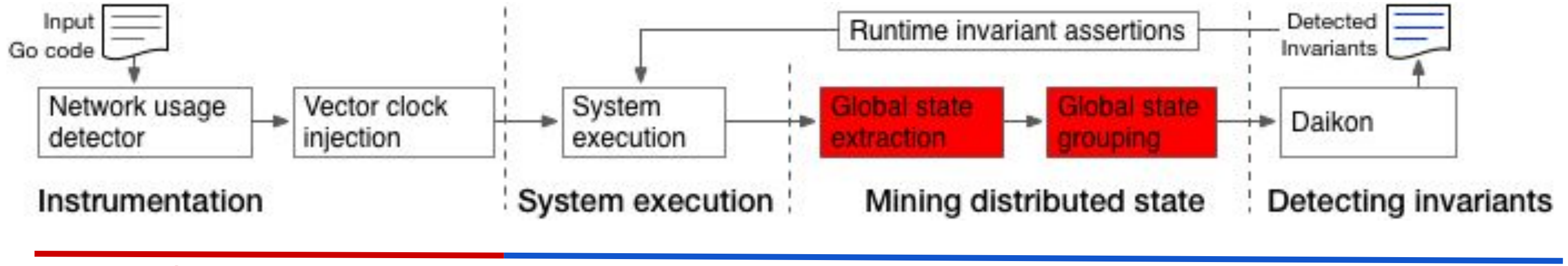
Log Collection (begin dynamic analysis)

- Analysis - performed on logs collected system execution
- Collection - execute a test suite on an instrumented system
- Quality - of Dinv’s invariants improve relative to test exhaustiveness



Dinv Overview

1. Distributed Invariant Inference Challenges
    a. What state should be logged and when?
    b. How to infer distributed invariants from logged state?
    c. How to enforce distributed invariants?



Static Analysis

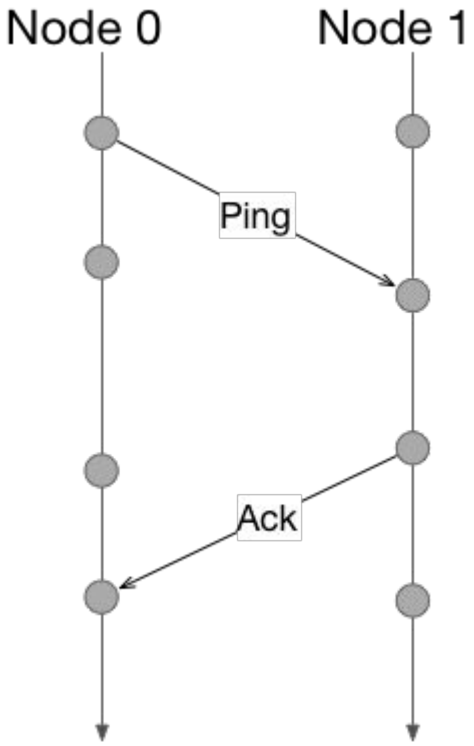
Dynamic Analysis

Consistent Cuts

Consistent cut: A partition of an execution, such that causality is preserved.

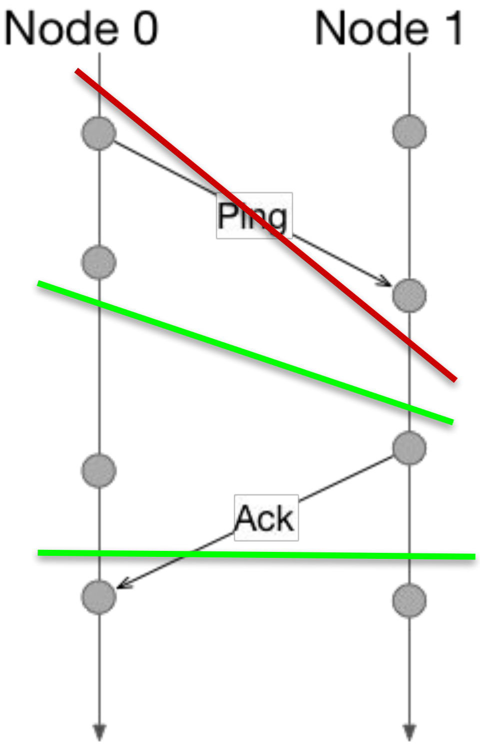
A consistent cut is a global observation of a distributed systems state

Example: Ping and Ack from Serf



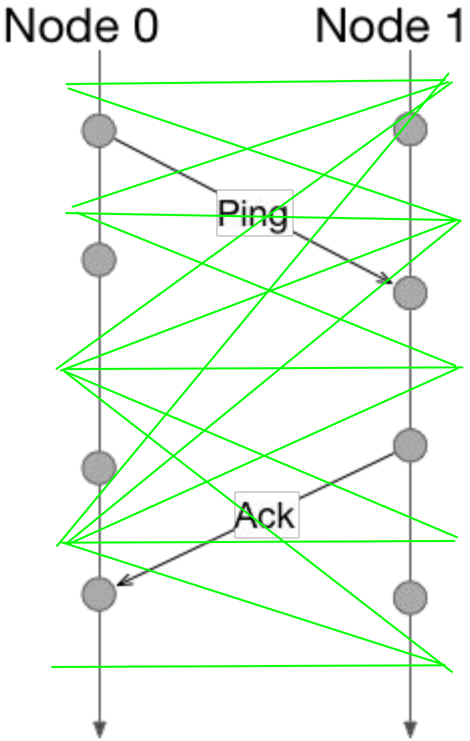
Consistent Cuts

- Green lines mark consistent cuts
  - No messages are in flight
  - Message is in flight
- The red line is not a consistent cut
  - The ping sent by Node 0 happened before the pings receipt on node 1.



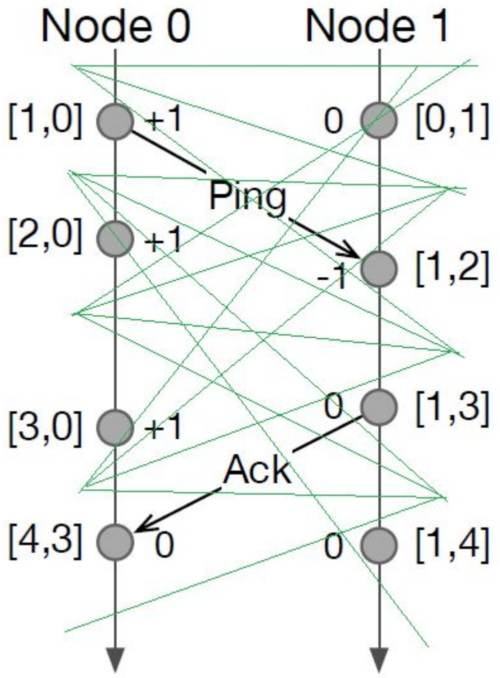
Consistent Cuts

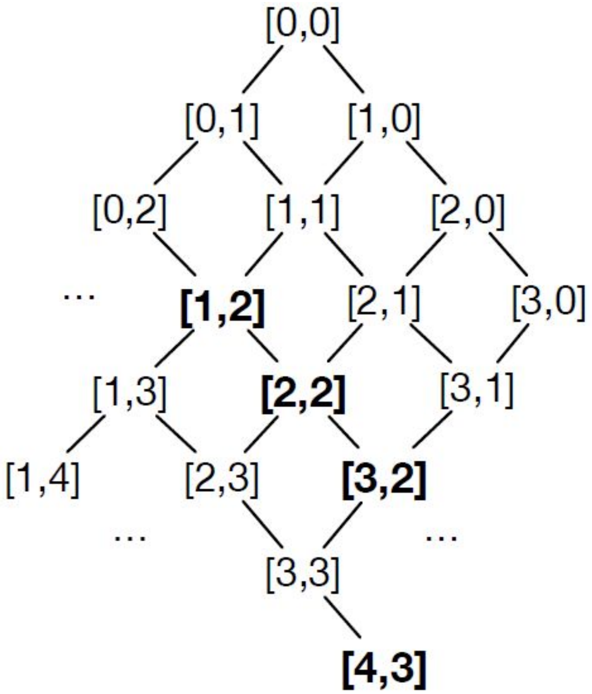
- Executions have an **exponential number** of consistent cuts
- Set of all consistent cuts compose every observable execution path.



Lattice Representation

- Cuts are naturally represented as a lattice
- Any path (downward), is a potential execution
- Trillions of points
- Exponentially more paths

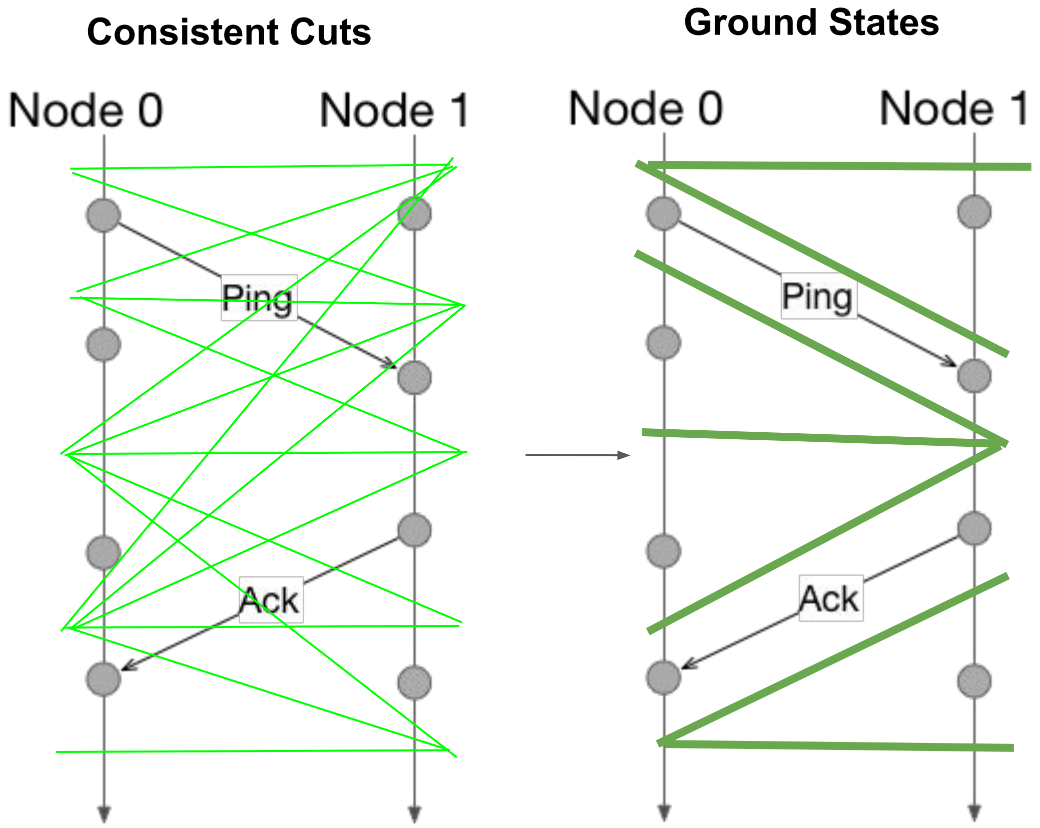




Corresponding **lattice**
(bold: no msgs in network**)**

Ground States

- Consistent cut is massive
- Require sampling heuristic
- **Ground States**: A consistent cut with no in flight messages
- Dramatically collapses search space



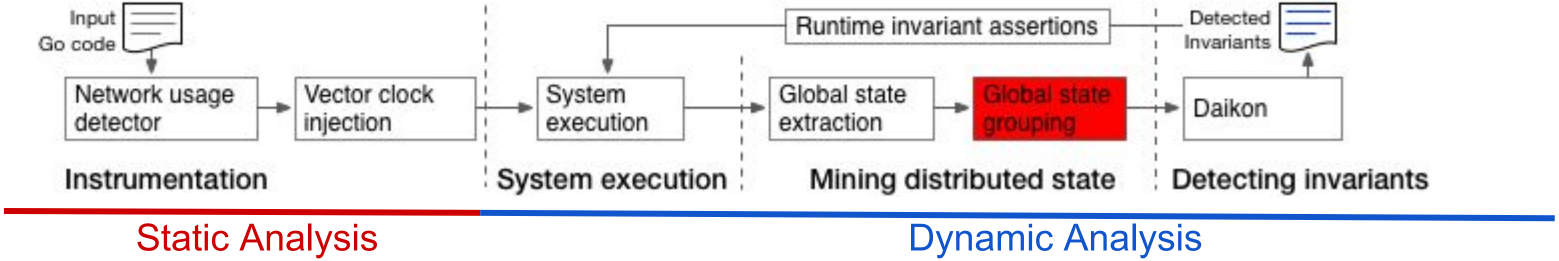
Consistent Cuts

Ground States

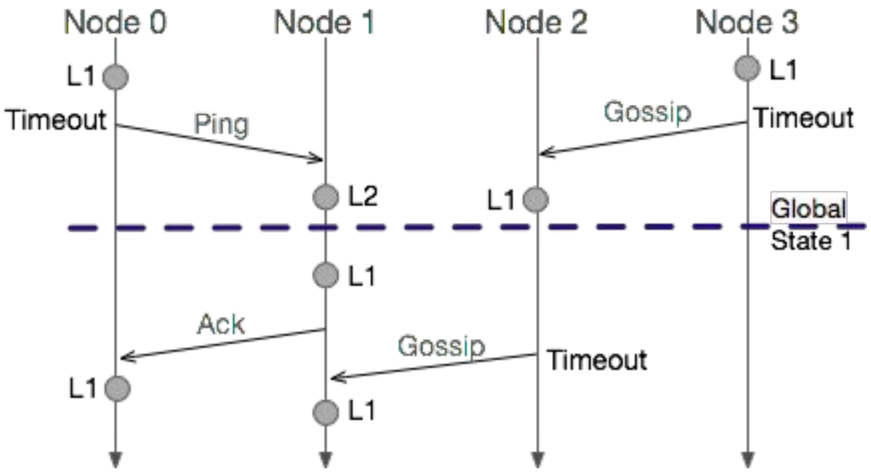
How to infer distributed invariants from global states

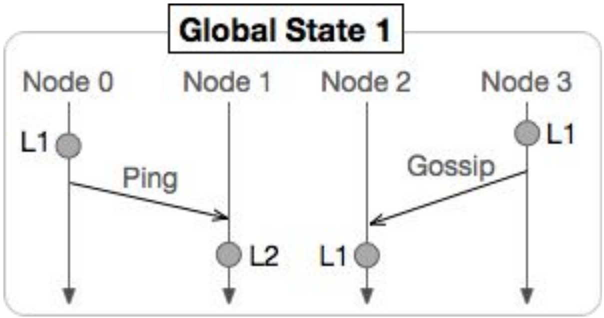
- Individual global states are a single instance in time
- Some invariants hold globally, others are protocol specific
- What state should be tested for invariants?

1) **All - States global merge**
2) Send-Receive - communication merge
3) Total order - transitive merge

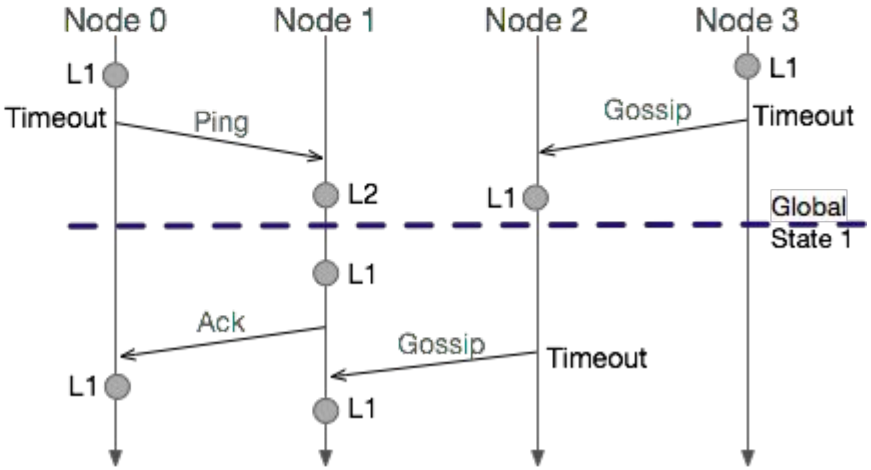


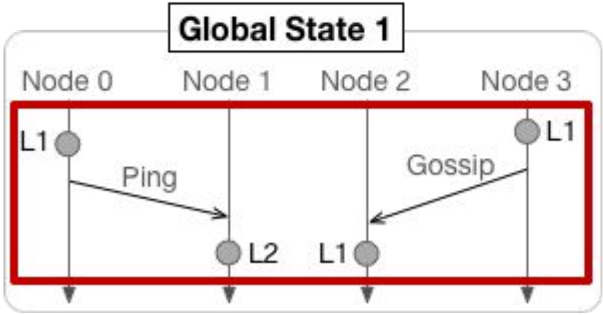
All-State Merging





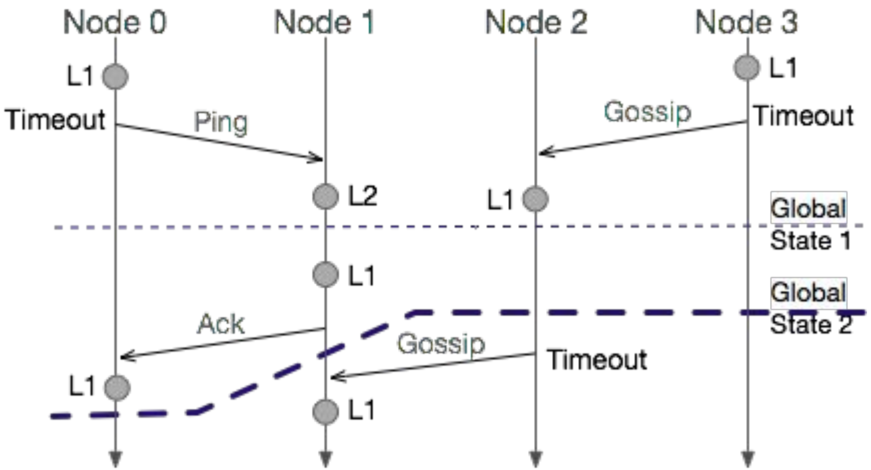
All-State Merging

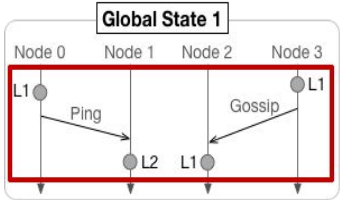




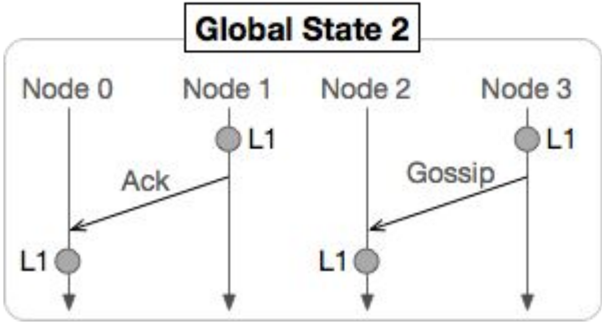
AS: [N0.L1,N1.L2, N2.L1, N3.L1]

All-State Merging

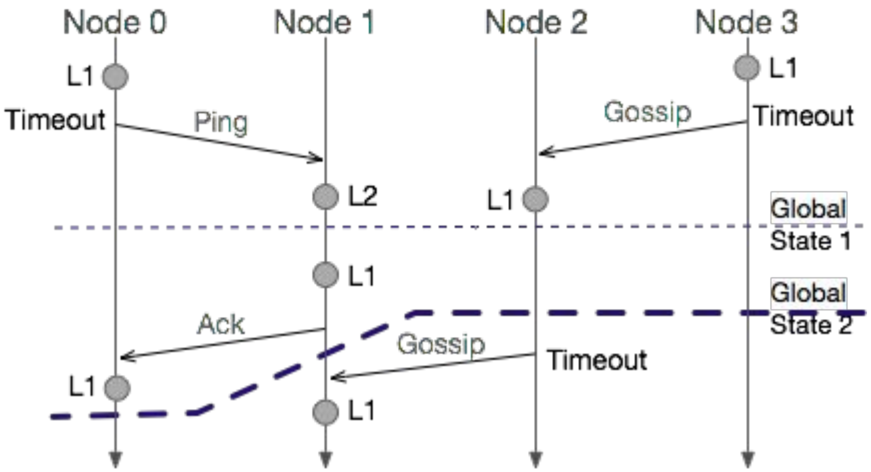


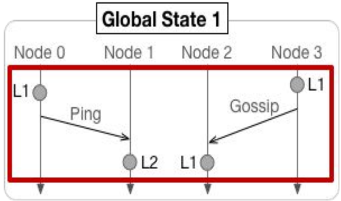


AS: [N0.L1,N1.L2, N2.L1, N3.L1]

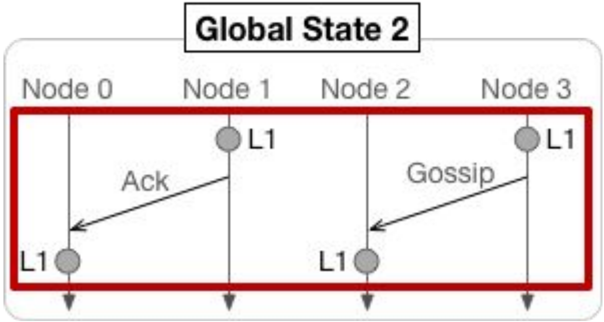


All-State Merging



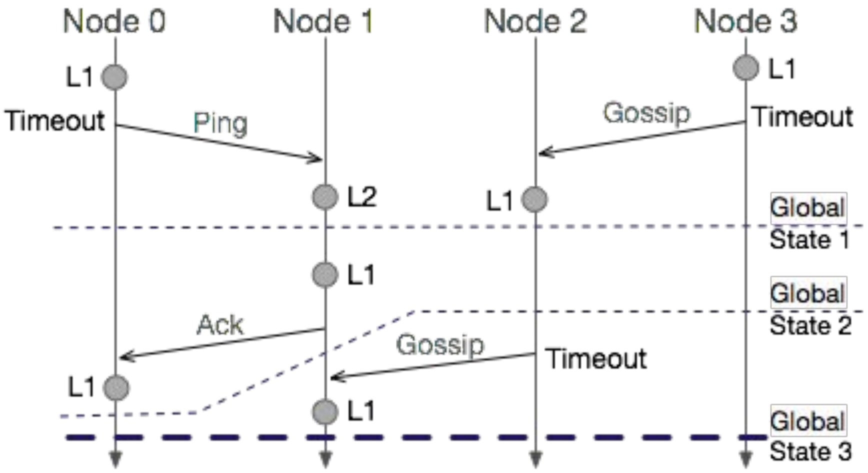


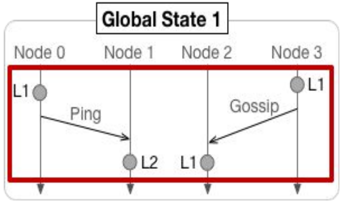
AS: [N0.L1,N1.L2, N2.L1, N3.L1]



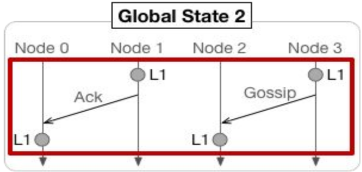
AS: [N0.L1,N1.L1, N2.L1, N3.L1]

All-State Merging

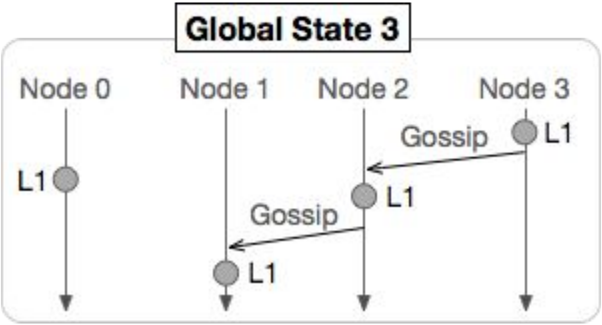




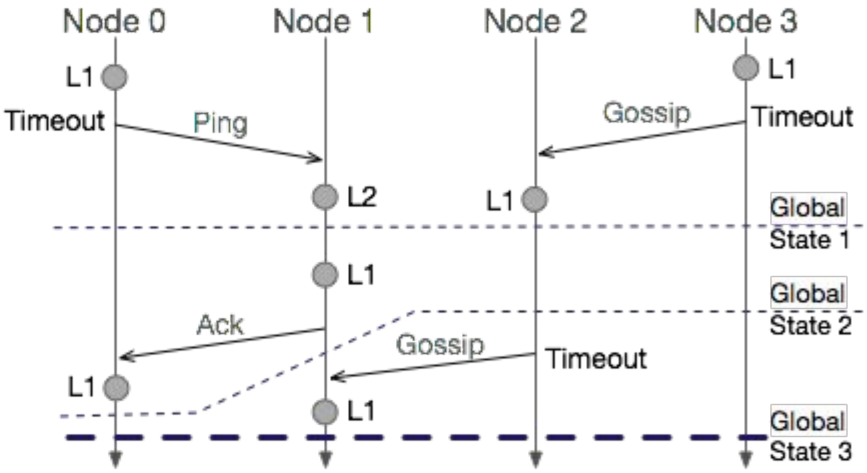
AS: [N0.L1,N1.L2, N2.L1, N3.L1]

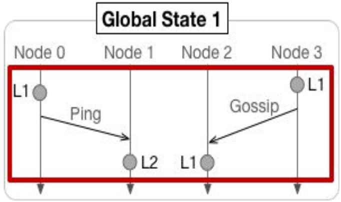


AS: [N0.L1,N1.L1, N2.L1, N3.L1]

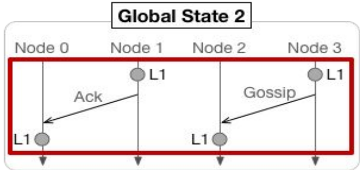


All-State Merging

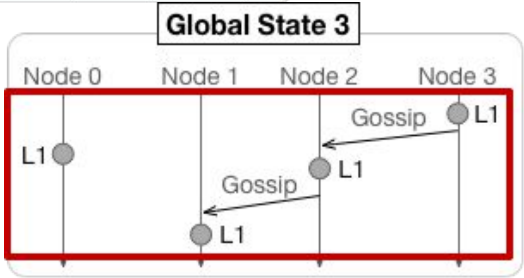




AS: [N0.L1,N1.L2, N2.L1, N3.L1]



AS: [N0.L1,N1.L1, N2.L1, N3.L1]

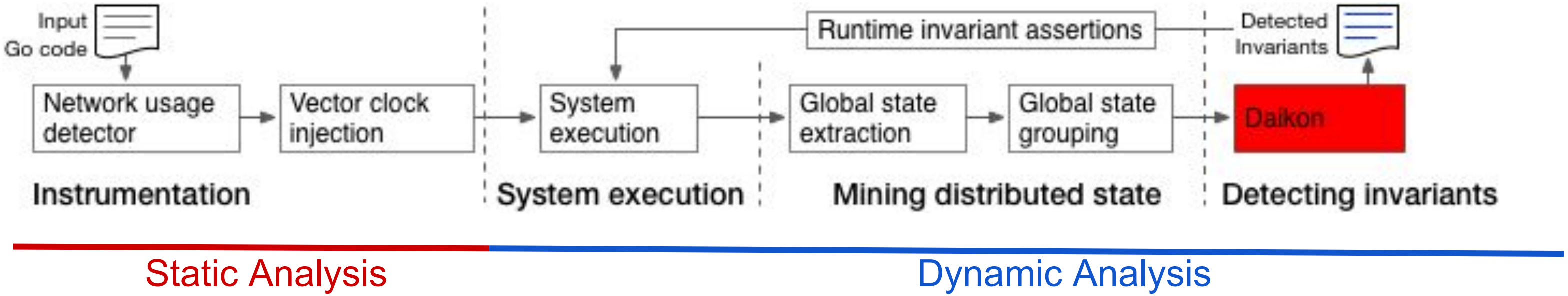


AS: [N0.L1,N1.L1, N2.L1, N3.L1]

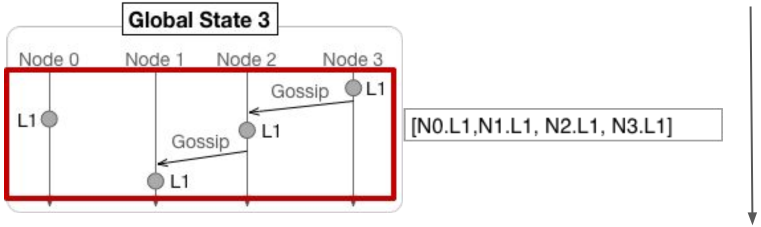
Inferring Invariants

- Daikon tool infers data invariants on data traces
- Insufficient for distributed systems (no partial order)

- Merged states are grouped by IDs and form serialized traces
- Extension for n-ary invariants

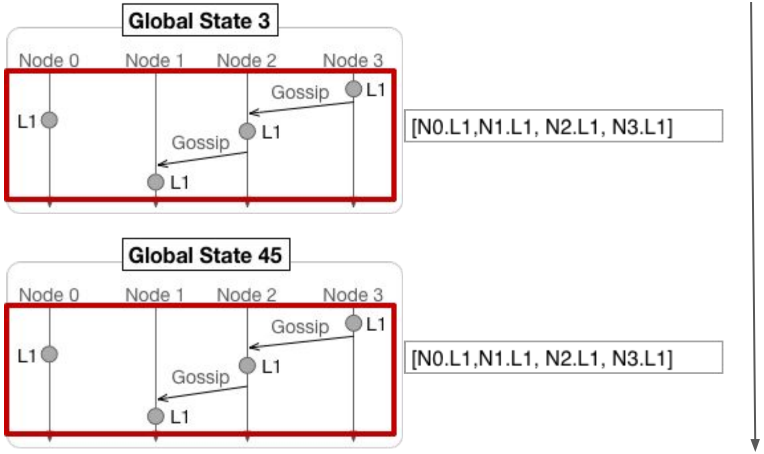


Daikon Bucketing



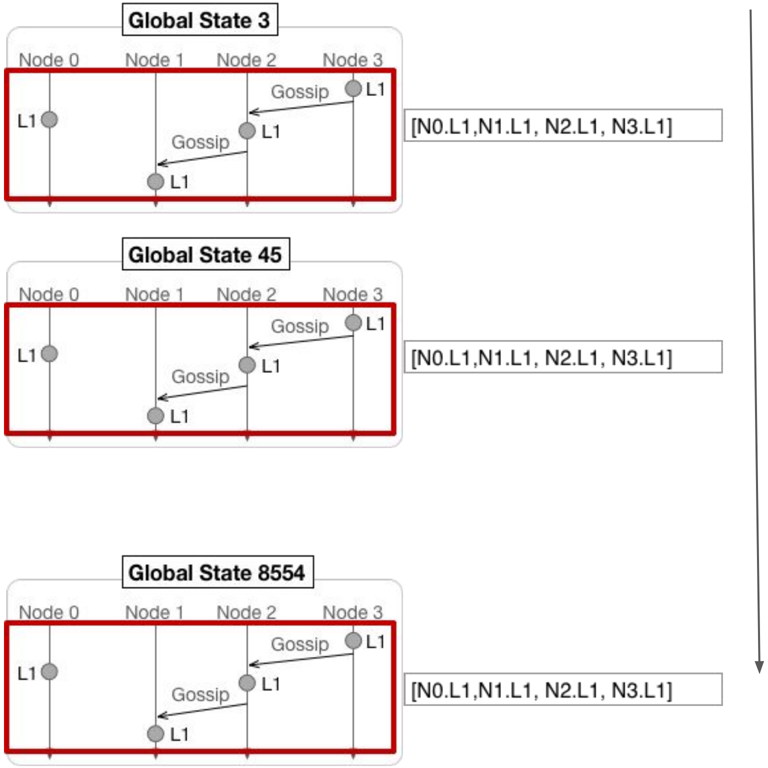
- Distributed States of the same signature are bucketed together

Daikon Bucketing



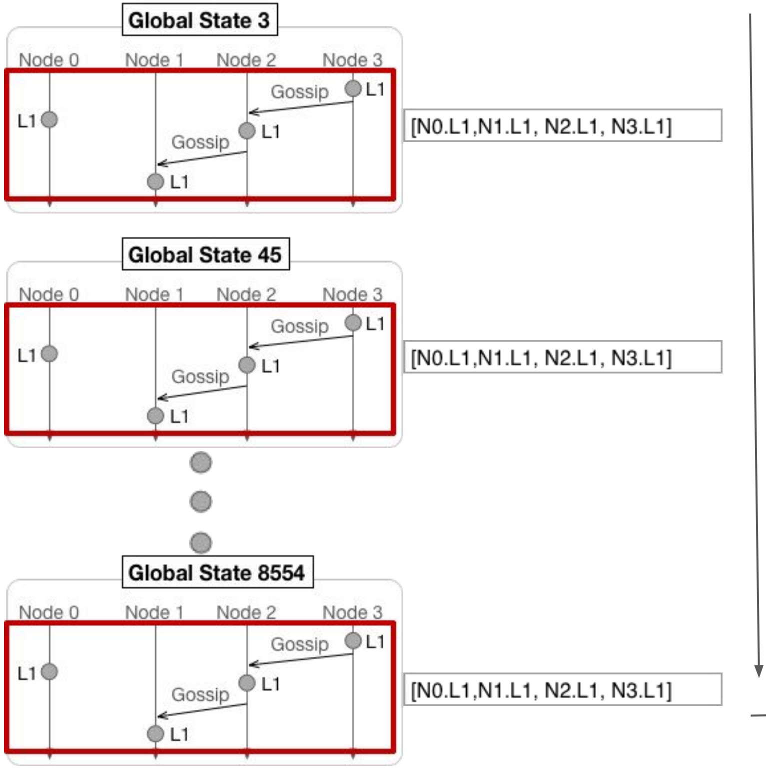
- Distributed States of the same signature are bucketed together
- More states means stronger invariants

Daikon Bucketing



- Distributed States of the same signature are bucketed together
- More states means stronger invariants
- Each new global state provides more evidence that an invariant is true

Daikon Bucketing



- Distributed States of the same signature are bucketed together
- More states means stronger invariants
- Each new global state provides more evidence that an invariant is true
- Ex) N1_Events == N2_Events == N3_Events

Daikon Template Invariant inference

Daikon systematically tests variables for invariants

Invariants are pre set in templates

Example operators)

==, >, >=, <, <=,

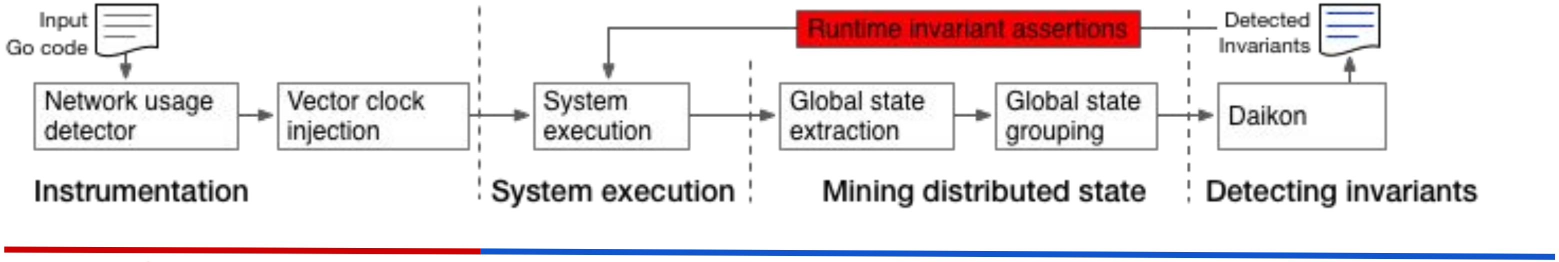
var1 + var2 = var3

Algorithm - Greater Than

```
For all ints i,j {
     If i > j {
          GreaterThan[i][j].Evidence++
     } Else {
          GreaterThan[i][j].Invariant = false
     }
}
```

Dinv Overview

1. Distributed Invariant Inference Challenges
    a. What state should be logged and when?
    b. How to infer distributed invariants from logged state?
    c. **How to enforce distributed invariants?**

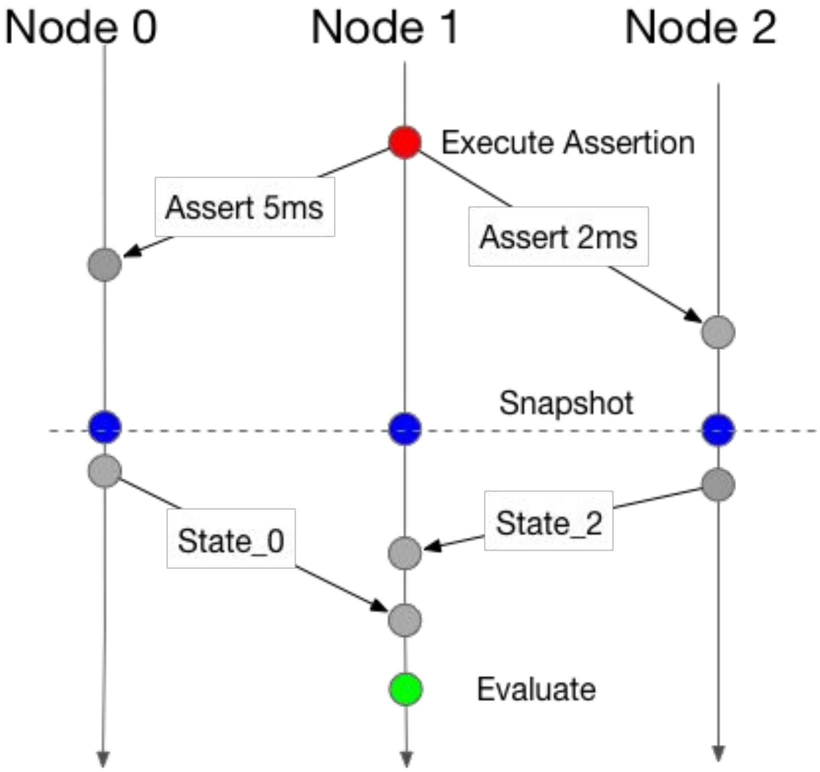


Static Analysis

Dynamic Analysis

Assertions

- Dinv has distributed assertions to enforce predicates at runtime
- See me after class if you want an overview!



Dinv Limitations

- Dinv’s dynamic analysis is incomplete
- Ground state sampling is poor on loosely coupled systems
- Temporal invariants are currently out of scope

Final take-aways

- Introduced dynamic and static analysis
- Discussed consistent+inconsistent cuts, distributed lattice, ground states
- Dinv tool for detecting data invariants in distributed systems + how it works:
  - Static identification of distributed state
  - Automatic static instrumentation
  - Post-execution merging of distributed states

Source code: https://bitbucket.org/bestchai/dinv

Demo: https://www.youtube.com/watch?v=n9fH9ABJ6S4