CPSC 416 - Distributed program analysis and Invariant Inference

March 28, 2018
Lecturer Stewart Grant
The problem: distributed systems are complex!

<table>
<thead>
<tr>
<th>Difficulty</th>
<th>Factors</th>
</tr>
</thead>
</table>
| 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)

Figure (1) A typical distributed systems developers desktop [my desktop]
What other techniques are available?

**Static analysis:**
Analyze a program's source code, without running the program.
- Type Checker
- Linter
- Symbolic Execution

*Complete but over-approximate and expensive.*

**Dynamic analysis:**
Analyze a program's 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 program's invariants is important for understanding if it is correct or faulty.

Example Program:

```python
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.

\[ \forall \text{Nodes } i, j, \text{ Keys}_i \neq \text{ Keys}_j \]
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 {
    // Code goes here
  }
  // More code
  // ...
```

Node 0  Node 1  Node 2  Node 3
Example Code: Serf

```go
1 func (s serfNode) serf(conn *UDPConnection) {
2     for true {
3         msg := conn.Read()
4         switch msg.Type {
5             case PING:
6                 conn.WriteToUDP("ACK", msg.Sender)
7                 break
8             
9         }
10        timeout := s.CheckForTimeouts()
11        switch timeout.Type {
12            case PING:
13                conn.WriteToUDP("PING", timeout.Node)
14                break
15                
16        }
17    }
18 }
19 }}
```
```go
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]

```go
func (s serfNode) serf(conn UDPEndPoint) {
    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)
        }
    }
}
```
What state should be logged and when?

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

Technique: Program slicing [Ottenstein 84]

- Transitivity track assignments to variables
- A slice is the complete set of statements over which marked data flows
What state should be logged and when?

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

Technique: Program slicing [Ottenstein 84]

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

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

<table>
<thead>
<tr>
<th>Instrumentation Strategy</th>
<th>Location choice</th>
<th>Variable Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function entrances/exits</td>
<td>Auto</td>
<td>Auto</td>
</tr>
<tr>
<td>Network calls</td>
<td>Auto</td>
<td>Auto</td>
</tr>
<tr>
<td>User-defined annotations</td>
<td>Manual</td>
<td>Manual or auto</td>
</tr>
</tbody>
</table>
How to Instrument with Dinv

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

Pre Instrumentation:

Two annotations:

@Track & @Dump

Track Recommended
(Reduces Output Size)
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
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
Example Vector Clock: Pack/Unpack

Pre-manual Instrumentation:

- Pack line 57
  Pack(payload interface{}) []byte

- Unpack line 64
  Unpack(buf []byte, toFill interface{})
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+0xa /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+0xa /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?
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
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

Node 0
Timeout
L1
Ping
L2
L1
Ack
Gossip
L1
Timeout
Node 1
Node 2
Node 3
L1
Gossip

Global State 1

AS: [N0.L1, N1.L2, N2.L1, N3.L1]
All-State Merging

AS: [N0.L1, N1.L2, N2.L1, N3.L1]
All-State Merging

<table>
<thead>
<tr>
<th>Action</th>
<th>Node</th>
<th>Node</th>
<th>Node</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ping</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Ack</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Gossip</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Timeout</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Global State 1

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

Global State 2

AS: [N0.L1, N1.L1, N2.L1, N3.L1]
All-State Merging

Global State 1

Node 0 Node 1 Node 2 Node 3
L1 Ping L2 L1 Gossip
L1 L1 L1

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

Global State 2

Node 0 Node 1 Node 2 Node 3
L1 Ack L1 L1 Gossip
L1 L1 L1

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

Global State 3

Node 0 Node 1 Node 2 Node 3
L1 Gossip L1 L1
L1 L1 L1

AS: [N0.L1, N1.L1, N2.L1, N3.L1]
All-State Merging

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?
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](https://bitbucket.org/bestchai/dinv)

Demo: [https://www.youtube.com/watch?v=n9fH9ABJ6S4](https://www.youtube.com/watch?v=n9fH9ABJ6S4)