NetCheck: Network Diagnoses from Blackbox Traces

Yanyan Zhuang*, Eleni Gessiou*, Fraida Fund*, Steven Portzer@, Monzur Muhammad^, Ivan Beschastnikh^, Justin Cappos*

(*) New York University, (^) University of British Columbia, (@) University of Washington
Goal

- Find bugs in networked applications
  - Large complex unknown applications
- Large complex unknown networks
- Understandable output / fix
Motivation

Chrome Client

Apache Server
Motivation

Chrome Client probing ping

Apache Server
Motivation

Chrome Client

probing

ping

Different traffic (ICMP)
Often different result

Apache Server
Motivation

Chrome Client

Apache Server
Motivation

Chrome Client

packet capture

Apache Server
Motivation

Requires detailed protocol / app knowledge

Chrome Client

Apache Server

packet capture
Motivation

Chrome Client

Apache Server
Motivation

Chrome Client

Model apps
Magpie, Xtrace, Pip...

Model

Apache Server
Motivation

Need a model per application

Model apps
Magpie, Xtrace, Pip...

Chrome Client

Model

Apache Server
Motivation

Chrome Client

Apache Server
Motivation

Chrome Client

Network Config Analysis

Header Space Analysis, etc.

Apache Server

Model & Config

Model & Config
Motivation

Need detailed network knowledge HW + config

Chrome Client

Network Config Analysis

Model & Config

Model & Config

Model & Config

Model & Config

Apache Server
Motivation

Chrome Client

Apache Server
NetCheck

Chrome Client

programmer

programmer

Apache Server

THE ARPA NETWORK
DEC 1969

4 NODES
Conceptual Sketch of Original Internet
NetCheck

Chrome Client

Model Programmer’s Understanding Deutsch’s Fallacies

programmer

programmer

Apache Server
Outline

- Motivation
- **NetCheck Overview**
- Trace Ordering
- Network Model
- Fault Classification
- Results / Conclusion
NetCheck overview

Application ➔ Traces ➔ NetCheck ➔ Likely Faults

Fail
NetCheck overview

Application

Traces

NetCheck

Likely Faults

ktrace

strace

Fail
NetCheck overview

Application

\[ \downarrow \]

Traces

\[ \downarrow \]

NetCheck

\[ \downarrow \]

 Likely Faults

Diagram:
- Host Traces
- Input
- Ordering Algorithm
  - syscall
  - simulation result
- Network Model
- Diagnoses Engine
  - simulation state errors
- Diagnosis
  - Output
NetCheck overview

Application

↓

Traces

↓

NetCheck

↓

 Likely Faults

Network Configuration Issues

Traffic Statistics

Problem Detected
Outline

- Motivation
- NetCheck Overview
- Trace Ordering
- Network Model
- Fault Classification
- Results / Conclusion
Traces

Series of locally ordered system calls
  Don’t want to modify apps or use a global clock
Gathered by strace, ktrace, systrace, truss, etc.
Call arguments and “return values”

socket() = 3
bind(3, ...) = 0
listen(3, 1) = 0
accept(3, ...) = 4
recv(4, "HTTP", ...) = 4
close(4) = 0

Call arguments
Return values
Return buffer
What we see is this:

Node A
1. socket() = 3
2. bind(3, ...) = 0
3. listen(3, 1) = 0
4. accept(3, ...) = 4
5. recv(4,"Hello", ..) = 5
6. close(4) = 0

Node B
1. socket() = 3
2. connect(3,...) = 0
3. send(3, "Hello",.) = 5
4. close(3) = 0

- one trace per host
- local order but no global order

Q: how do we reconstruct what really happened?
What we want is this

A1. socket() = 3
B1. socket() = 3
A2. bind(3, .. .) = 0
A3. listen(3, 1) = 0
B2. connect(3,...) = 0
A4. accept(3, ...) = 4
B3. send(3, "Hello", ...) = 5
A5. recv(4, "Hello", ...) = 5
B4. close(3) = 0
A6. close(4) = 0
A1. socket() = 3  
B1. socket() = 3  
A2. bind(3, .. .) = 0  
A3. listen(3, 1) = 0  
B2. connect(3,...) = 0  
A4. accept(3, ...) = 4  
B3. send(3, "Hello", ...) = 5  
A5. recv(4, "Hello", ...) = 5  
B4. close(3) = 0  
A6. close(4) = 0

What we want is this

The ground truth

Goal: find an equivalent interleaving
Observation 1: Order Equivalence

Node A
1. socket() = 3
2. bind(3, ...) = 0
3. listen(3, 1) = 0
4. accept(3, ...) = 4
5. recv(4,"Hello", ..) = 5
6. close(4) = 0

Node B
1. socket() = 3
2. connect(3,..) = 0
3. send(3, "Hello",..) = 5
4. close(3) = 0

- one trace per host
- local order but no global order
Q: how do we reconstruct what really happened?
The socket() calls are not visible to the other side
Some orders are equivalent!
Observation 2: Return Values Guide
Ordering

Node A
1. socket() = 3
2. bind(3, ...) = 0
3. listen(3, 1) = 0
4. accept(3, ...) = 4
5. recv(4,"Hello", ..) = 5
6. close(4) = 0

Node B
1. socket() = 3
2. connect(3,...) = 0
3. send(3, "Hello",.) = 5
4. close(3) = 0

- one trace per host
- local order but no global order
Q: how do we reconstruct what really happened?
A call’s return value **may-depend-on** a remote call’s action

**Result indicates order of calls**
Deciding call order

full set of may-depend-on relations
Ordering Algorithm

Input traces

Algorithm process

Output Ordering

Host A trace:
A1. socket(...) = 4
A2. bind(4, ...) = 0
A3. listen(4, 1) = 0
A4. accept(4, ...) = 6
A5. recv(6, "Hello", ...) = 5

Host B trace:
B1. socket(...) = 3
B2. connect(3, ...) = 0
B3. send(3, "Hello", ...) = 5
Ordering Algorithm

Input traces

A

socket
bind
listen
accept
recv

B

socket
connect
send

Algorithm process

Try socket on host A: accepted

Output Ordering

A

socket

Host A trace:
A1. socket(...) = 4
A2. bind(4, ...) = 0
A3. listen(4, 1) = 0
A4. accept(4, ...) = 6
A5. recv(6, "Hello", ...) = 5

Host B trace:
B1. socket(...) = 3
B2. connect(3, ...) = 0
B3. send(3, "Hello", ...) = 5
Ordering Algorithm

Input traces

A
listen

accept
recv

B
connect

send

Algorithm process

Try connect on host B: rejected

Output Ordering

A
socket

B
socket

A
bind

Host A trace:
A1. socket(...) = 4
A2. bind(4, ...) = 0
A3. listen(4, 1) = 0
A4. accept(4, ...) = 6
A5. recv(6, "Hello", ...) = 5

Host B trace:
B1. socket(...) = 3
B2. connect(3, ...) = 0
B3. send(3, "Hello", ...) = 5
Ordering Algorithm

Input traces

A
- listen
- accept
- recv

B
- connect
- send

Algorithm process

Try listen on host A: accepted

Output Ordering

A
- socket
B
- socket
- listen

Host A trace:
A1. socket(...) = 4
A2. bind(4, ...) = 0
A3. listen(4, 1) = 0
A4. accept(4, ...) = 6
A5. recv(6, "Holal", ...) = 5

Host B trace:
B1. socket(...) = 3
B2. connect(3, ...) = 0
B3. send(3, "Hello", ...) = 5
Ordering Algorithm

Input traces

A
recv

B
send

Algorithm process

Try recv on host A: rejected

Output Ordering

TCP BUFFER: “”

socket

A
socket

bind

A
listen

B
connect

A
accept

Host A trace:
A1. socket(...) = 4
A2. bind(4, ...) = 0
A3. listen(4, 1) = 0
A4. accept("") = 6
A5. recv(6, "Hola!", ...) = 5

Host B trace:
B1. socket(...) = 3
B2. connect(3, ...) = 0
B3. send(3, "Hello", ...) = 5
Ordering Algorithm

Input traces
A
recv
B
send

TCP BUFFER: ""

Algorithm process
Try send on host B: accepted

Output Ordering

A
socket
B
socket
A
bind
B
listen
A
connect
B
accept
A
send
Ordering Algorithm

Input traces
A
recv
B
None

TCP BUFFER: “Hello”

Algorithm process
Try send on host B: accepted

Output Ordering

socket

bind

listen

connect

accept

send

Host A trace:
A1. socket(...) = 4
A2. bind(4, ...) = 0
A3. listen(4, 1) = 0
A4. accept(...) = 6
A5. recv(6, "Hola!") = 5

Host B trace:
B1. socket(...) = 3
B2. connect(3, ...) = 0
B3. send(3, "Hello", ...) = 5
Ordering Algorithm

Input traces
A
B
recv
recv

Algorithm process
Try recv on host A:
Fatal Error

Output Ordering

TCP BUFFER: “Hello”

A
socket
B
socket
A
bind
B
listen
A
connect
B
accept
A
send

Host A trace:
A1. socket(...) = 4
A2. bind(4, ...) = 0
A3. listen(4, 1) = 0
A4. accept(...) = 6
A5. recv(6, “Hola!” ...) = 5

Host B trace:
B1. socket(...) = 3
B2. connect(3, ...) = 0
B3. send(3, "Hello", ...) = 5
Outline

- Motivation
- NetCheck Overview
- Trace Ordering
- **Network Model**
- Fault Classification
- Results / Conclusion
Network Model

- Simulates invocation of a syscall
  - datagrams sent/lost
    - reordering / duplication is notable
  - track pending connections
  - buffer lengths and contents
    - send -> put data into buffer
    - recv -> pop data from buffer

- Simulation outcome
  - *Accept* → can process (correct buffer)
  - *Reject* → wrong order (incomplete buffer)
  - *Permanent reject* → abnormal behavior (incorrect buffer)
Network Model

- Simulates invocation of a syscall
- Capture programmer assumptions
  - Assumes a simplified network view
    - Assume transitive connectivity
  - Little, random loss
  - No middle boxes
- Assume uniform platform
  - Flag OS differences
How Model Return Values Impact Trace Ordering

- Blackbox Tracing mechanism

Trace Ordering: linear running time (total trace length) * number of traces
Outline

• Motivation
• NetCheck Overview
• Trace Ordering
• Network Model
• Fault Classification
• Results / Conclusion
Fault Classifier

- Goal: Decide what to output
- Problem: Show relevant information
- Fault classifier: global (rather than local) view
  - uncovers high-level patterns by extracting low-level features
  - Examples: middleboxes, non-transitive connectivity, MTU, mobility, network disconnection
  - All look like loss, but have different patterns in the context of other flows
Fault Classifier

- Options to show different levels of detail
- Network admins / developers
  - detailed info
- End users
- Classification
- Recommendations
Outline

• Motivation
• NetCheck Overview
• Trace Ordering
• Network Model
• Fault Classification
• Results / Conclusion
Evaluation: Production Application Bugs

- Reproduce reported bugs from bug trackers (Python, Apache, Ruby, Firefox, etc.)
  - A total of 71 bugs
  - Grouped into 23 categories
    - Virtualization incurred/portability bugs
    - SO_REUSEADDR behaves differently across OSes
    - accept inherit O_NONBLOCK
    - ...
  - Correct analysis of >95% bugs
Evaluation: Observed Network Faults

- Twenty faults observed in practice on a live network
  - MTU bug
    - Intermediary device
  - Port forward
    - Traffic sent to non-relevant addresses
  - Provide supplemental info
    - packet loss
    - buffers being closed with data in
  - 90% of cases correctly detected
General Findings in Practice

- **Middle boxes**
  - Multiple unaccepted connections
    - client behind NAT in FTP
- **TCP/UDP**
  - non-transitive connectivity in VLC
- **Complex failures**
  - VirtualBox send data larger than buffer size
  - Pidgin returned IP different from bind
  - Skype NAT + close socket from a different thread
- **Used on Seattle Testbed** [seattle.poly.edu]
NetCheck Performance Overhead

![Graph showing performance overhead for different applications such as Skype, Firefox, VLC, Telnet, and SSH.][1]

---

[1]: #/image-url
Conclusion

Built and evaluated NetCheck, a tool to diagnose network failures in complex apps

● Key insights:
  ○ model the programmer’s misconceptions
  ○ relation between calls → reconstruct order

● NetCheck is effective
  ○ Everyday applications & networks
  ○ Real network / application bugs
  ○ No per-network knowledge
  ○ No per-application knowledge

Try it here: https://netcheck.poly.edu/
Backup slides.
What is NetCheck?

- No app- or network-specific knowledge
- No modification to apps/infrastructure
- No synchronized global clock

- Blackbox Tracing mechanism (eg, strace)
  - Reconstruct a plausible total ordering of syscall traces from multiple hosts
  - Uses simulation and captured state to identify network related issues
  - Map low-level issues to higher-level characterizations of failure
Diagnosis Model

- Blackbox Tracing mechanism

![Diagram showing the flow of Traces through Trace Ordering, Call dependency, Application-Agnostic Model, and Collating Fault Classifier.]

Call dependency
Diagnosis Model

- Blackbox Tracing mechanism

![Diagram showing the call dependency and accept/reject/FE processes between traces, trace ordering, collating fault classifier, and application-agnostic model.](image)
Diagnosis Model

- Blackbox Tracing mechanism

Trace Ordering: linear running time

Trace Ordering

Call dependency
accept/reject/FE

Application-Agnostic Model

Collating Fault Classifier

reject → reorder
Pseudocode and Analysis

1. push trace $t_0$ in stack $s_0$, ..., trace $t_{n-1}$ in stack $s_{n-1}$
2. while $(s_0, \ldots, s_{n-1})$ not empty:
3. \hspace{1em} q = \text{peek\_stack}(s_0, \ldots, s_{n-1}); \hspace{1em} q\text{.sort}(\text{priority})
4. \hspace{1em} while True:
5. \hspace{2em} if q empty: \hspace{1em} raise FatalError
6. \hspace{2em} i, j = q\text{.dequeue}();
7. \hspace{1em} outcome = \text{model\_simulate}(i, j)
8. \hspace{1em} if outcome == ACCEPT:
9. \hspace{2em} ordered_trace.push(s_j.pop()); \hspace{1em} break
10. \hspace{1em} elif outcome == REJECT: \hspace{1em} pass
11. \hspace{1em} elif outcome == FatalError: \hspace{1em} raise FatalError

**Pseudocode Analysis**

Best case: $O(1)$
Worst case: $O(n)$

**Overall:**
Best case $O(L)$
Worst Case $O(n*L)$
1. push trace $t_0$ in list $s_0$, ..., trace $t_{n-1}$ in list $s_{n-1}$
2. while ($s_0$, ..., $s_{n-1}$) not empty:
3.   $q = \text{peek\_stack}(s_0, \ldots, s_{n-1});$  \hspace{1cm} $q.sort(\text{priority})$
4.   while True:
5.     if $q$ empty:   raise FatalError
6.     $i_j = q.\text{dequeue}();$
7.     outcome = model\_simulate($i_j$)
8.     if outcome == ACCEPT:     \hspace{1.5cm} Accept $\rightarrow$ Traverse
9.         ordered_trace.\text{push}(s_j.\text{pop}());  \hspace{1cm} break
10.    elif outcome == REJECT: \hspace{1cm} Reject $\rightarrow$ Backtrack
11.       continue
12.    elif outcome == FatalError:  raise FatalError
NetCheck input

Node A
1. `socket() = 3`
2. `bind(3, ...) = 0`
3. `listen(3, 1) = 0`
4. `accept(3, ...) = 4`
5. `recv(4, "Hello", ..) = 5`
6. `close(4) = 0`

Node B
1. `socket() = 3`
2. `connect(3, ...) = 0`
3. `send(3, "Hello", ..) = 5`
4. `close(3) = 0`
## NetCheck input

<table>
<thead>
<tr>
<th>Node A</th>
<th>Syscall</th>
<th>Node B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. socket()</td>
<td>3</td>
<td>1. socket()</td>
</tr>
<tr>
<td>2. bind(3, ...)</td>
<td>0</td>
<td>2. connect(3, ...)</td>
</tr>
<tr>
<td>3. listen(3, 1)</td>
<td>0</td>
<td>3. send(3, &quot;Hello&quot;, ..) = 5</td>
</tr>
<tr>
<td>4. accept(3, ...)</td>
<td>4</td>
<td>4. close(3)</td>
</tr>
<tr>
<td>5. recv(4, &quot;Hello&quot;, ..)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>● 6. close(4)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
connect depends on listen

Order 1
A1 bind(3, ...) = 0
A2 listen(3, 5) = 0
B1 connect(3, ...) = 0

Order 2
A1 bind(3, ...) = 0
B1 connect(3, ...) = -1 ECONNREFUSED
A2 listen(3, 5) = 0

Order 3
B1 connect(3, ...) = -1 ECONNREFUSED
A1 bind(3, ...) = 0
A2 listen(3, 5) = 0
Example Rules

- **Middle boxes**
  - Multiple unaccepted connections  \( \Rightarrow \) client behind NAT in FTP
  - Missing connect on accepted connections  \( \Rightarrow \) server behind NAT or port forwarding
  - Multiple connect non-standard failure  \( \Rightarrow \) firewall filtering connections
  - Multiple connect to listening address get refused
  - Multiple non-blocking connect failure
  - Traffic sent to non-relevant addresses  \( \Rightarrow \) NAT or 3rd party proxy/traffic forwarding
Example fault classifier rules

● Middle boxes
  ○ Multiple unaccepted connections
    $\Rightarrow$ client behind NAT in FTP
  ○ Missing connect on accepted connections $\rightarrow$ server behind NAT or port forwarding
  ○ Traffic sent to non-relevant addresses $\rightarrow$ NAT or 3rd party proxy/traffic forwarding

● TCP
  ○ select/poll timeout
  ○ send data after connection closed
Example rules (cont.)

- **UDP**
  - datagram sent/lost per connection
  - high datagram loss rate
    - \( \Rightarrow \) *non-transitive connectivity in VLC*

- **Misc**
  - apps send data larger than default OS buffer size
    - \( \Rightarrow \) *bug report from VirtualBox bug tracker*
  - returned IP different from bind
    - \( \Rightarrow \) *simultaneous net disconnect/reconnect in Pidgin*
  - **Skype** attempted to close socket from a different thread
Evaluation: Everyday Applications

- **FTP**
  - All reverse connections from server lost
    - Client behind NAT
- **Pidgin**
  - getsockname returns different IP
    - Client poor connection results in IP changes
- **Skype**
  - Poor call quality, msg drop
    - Network delay, NAT
    - Skype closes socket from different thread
- **VLC**
  - Packet loss
    - Non-transitive connectivity issue