

Verteilte Systeme

Verteiltes Debugging

Fehlerkategorien und –arten bei der verteilten Ausführung



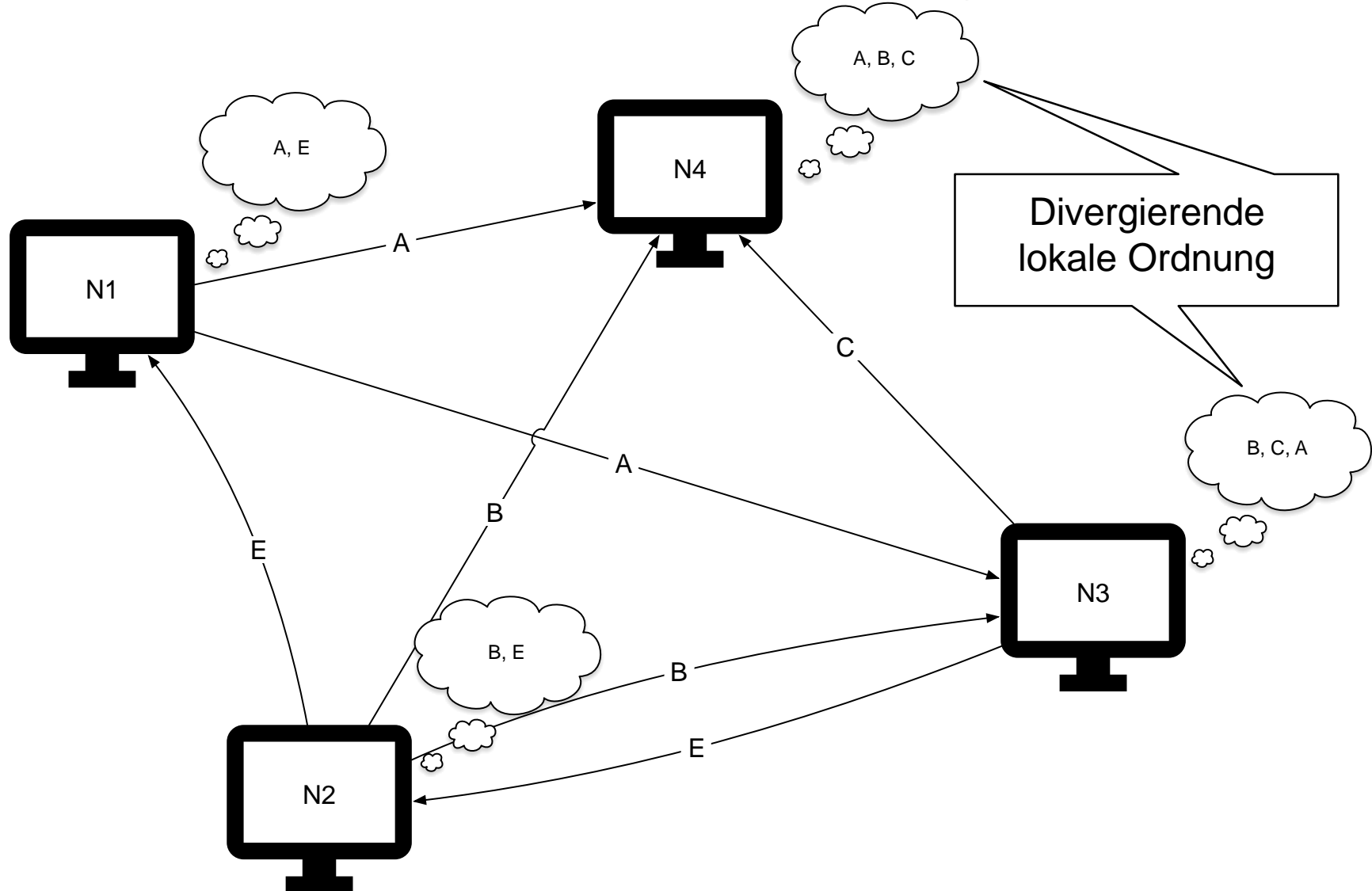
Warum verteilte Ausführung?

- ◆ **Skalierbarkeit:**
 - **Hohe Anfrage** an einen Dienst **übersteigt die Ressourcen** einer einzelnen Maschine
 - **Speicherbedarf** vieler Anwendungen (siehe Google, Facebook, etc.) übersteigt das Maximum einzelner Server
 - **Software muss** mit der Anzahl an Benutzern „**mitwachsen**“ können, um marktgerecht zu sein
- ◆ **Robustheit:**
 - Der **Ausfall** einzelner Komponenten muss **kompensierbar** werden
 - Infrastruktur-Software muss **hochverfügbar** sein **durch Redundanzen und Fehlertoleranz**

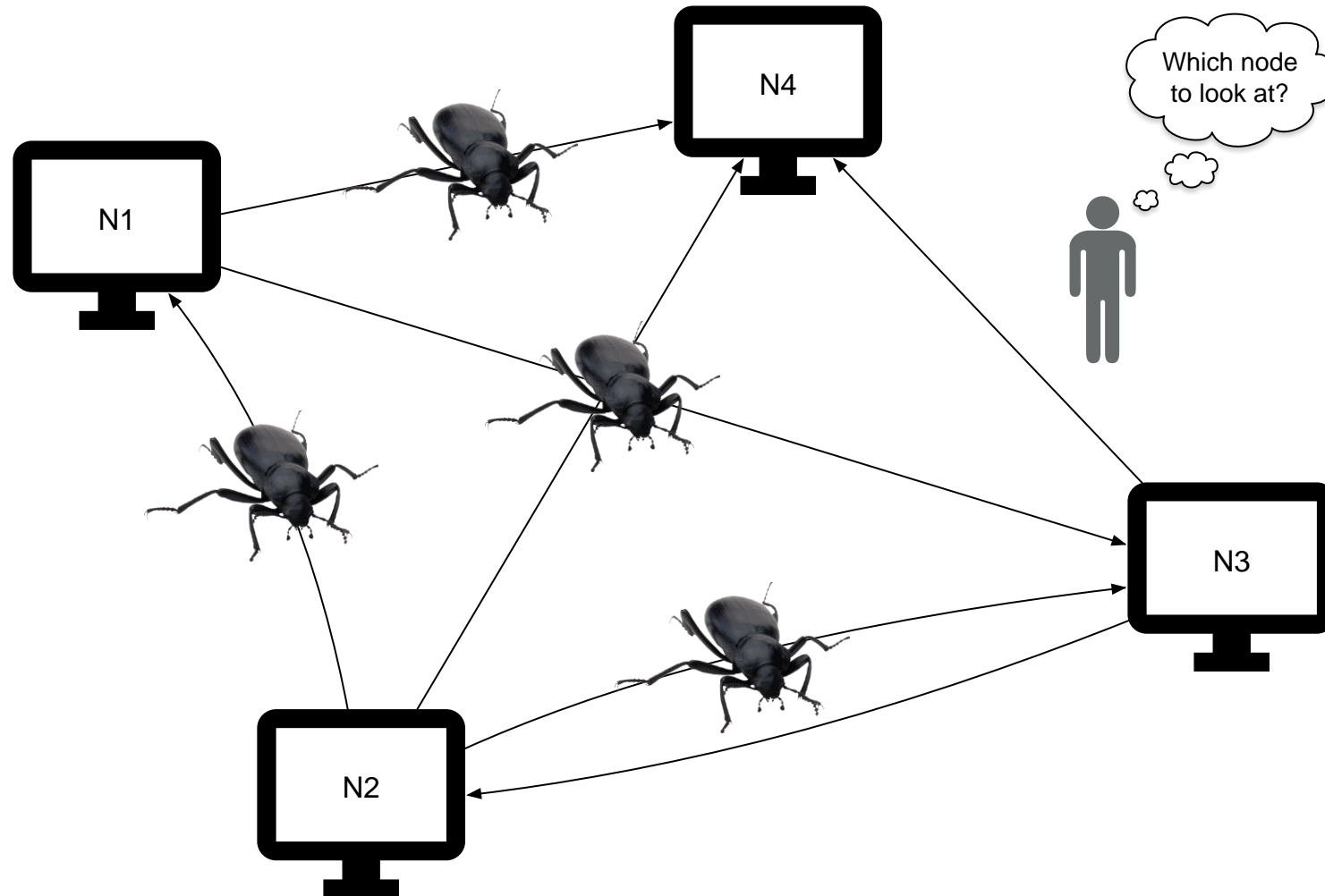
Eigenschaften verteilter Ausführung

- ◆ **Lokale Ordnung:**
 - Kein Knoten sieht alle Ereignisse
 - Latenz und Topologie entscheiden Ereignisreihenfolge
 - Zeitliche Ordnung impliziert *nicht* Kausalität
- ◆ **Nichtdeterminismus:**
 - Scheduler entscheidet Verarbeitungsabläufe dynamisch
 - Unterschiedliche Laufzeiten gleicher Aufgaben
 - Mehrfache Programmdurchläufe können unterschiedliche Ereignisketten produzieren
- ◆ **Teilausfälle:**
 - **Hardware fällt häufig aus** in großen verteilten Systemen:
 - 10.000 Server haben ständig Ausfälle
 - Software muss **Fehler** der vorhandenen Hardware **antizipieren** und kritische Komponenten **replizieren**

Verteilte Ausführung



Fehlersuche in verteilten Systemen



Fehlerarten

- ◆ Fehler bei **regulärem Betrieb**:
 - **Verfügbarkeit**: System fällt durch Programmfehler partiell oder vollständig aus
 - **Logik**: Anwendung erzeugt ungültige oder fehlerhafte Ergebnisse, läuft aber ununterbrochen weiter im Fehlerfall
 - **Synchronisation**: Komponenten divergieren in ihrer Sicht auf (globale) Systemzustände und produzieren widersprüchliche Ergebnisse
- ◆ Fehler **unter Last**:
 - **Skalierbarkeit**: Software nutzt vorhandene Rechenkapazitäten unzureichend und kann Anfragen nicht zeitgerecht bearbeiten
 - **Stabilität**: Ausfallrate steigt mit der Anzahl der Teilnehmer im System bis hin zum Totalausfall

Debugging



Debugging

- ◆ Prozess zum **Auffinden von Fehlerursachen**, deren Symptome sich als fehlerhaftes Programmverhalten (*Bugs*) äußern, z.B. falsche Ergebnisse, Dead-/Lifelocks, Programmabstürze, etc.
- ◆ Typischer Ablauf:
 - Auftreten eines **Fehlers im Produktiv- oder Testeinsatz**
 - **Spurensuche** nach der Fehlerursache
 - **Reproduktion** des Fehlers
 - **Lokalisierung** der Ursache, z.B. durch minimierte Tests
 - **Beheben** der Fehlerursache

Debugger

- ◆ Werkzeug zur **methodischen Analyse** eines Programmes
- ◆ **Kontrollieren** des Programmablaufes
 - Haltepunkte in kritischen Code-Pfaden
 - Einzelschritt-Verarbeitung
- ◆ **Inspizieren** des Zustands einer laufenden Anwendung
 - RAM: Speicherverbrauch und Inhalt Heap-allokierter Daten
 - Register und Stack: Variablen innerhalb der aktuellen Funktion und aller aufrufenden Funktionen
- ◆ **Modifizieren** von Zustand und Code
 - Überschreiben von Speicherinhalten
 - Quellcode-Änderung in laufenden Programmen (*just in time debugging*)

Debugging verteilter Anwendungen

- ◆ **Kernunterschiede** zu klassischem Debugging:
 - Kein einheitlicher, gemeinsamer Speicher
 - Keine einheitliche Zeit
 - Keine globale Ordnung von Ereignissen
- ◆ **Fehler** sind oft **schwierig zu reproduzieren**:
 - Programmablauf über viele Maschinen verteilt
 - Lokal sichtbare Reihenfolge von Ereignisketten kann sich mit jedem Durchlauf ändern
 - Netzwerkkonfiguration und -laufzeiten beeinflussen Programmablauf, sind aber u.U. nicht oder nur schwer in Testaufbauten nachstellbar

Besonderheiten verteilter Software

- ◆ **Fehlerwahrscheinlichkeit proportional zur Systemgröße**
 - Netzwerkfehler (Hardware/Software/Fehlkonfiguration/...)
 - Hardwareausfälle (Stromausfall/Defekt/Wartung/...)
- ◆ **Fehlerbehandlung kann kein Nachgedanke sein**
 - Robustheit muss von Anfang an mitgedacht werden
 - Fehlerbehandlung ist kritischer, häufig laufender Code (Unit Tests für Fehler!)
- ◆ **Partielle Fehler** machen Debugging komplexer
 - Nachstellen von Fehlern oft schwierig (z.B. Simulieren von kurzzeitigen Teilausfällen im Netzwerk)
 - Fehlerursachen oft im Zusammenspiel mehrerer Schichten
- ◆ Logik für Funktionalität und Fehlerbehandlung **eng verzahnt**
 - Konsensalgorithmen müssen robust gegen Ausfälle sein
 - Datenbanken müssen konsistent bei Teilausfällen bleiben

Grenzen von Debuggern

- ◆ **Annahmen von „klassischen“ Debuggern (z.B. GDB):**
 - Globale Kontrolle über Speicher und Ausführung
 - Konsistenter Zustand aller Systemteile (alle Threads sehen den gleichen Speicherinhalt)
- ◆ **Probleme bei Einsatz von Debuggern in verteilten Systemen:**
 - *Zeitverhalten*: Timeouts und variierende Kausalitätsketten
 - *Lokalität*: unabhängige Speicherbereiche / Variablen
 - *Nichtdeterminismus*: globaler Zustand nicht bestimmbar

Rolle des Programmierparadigmas

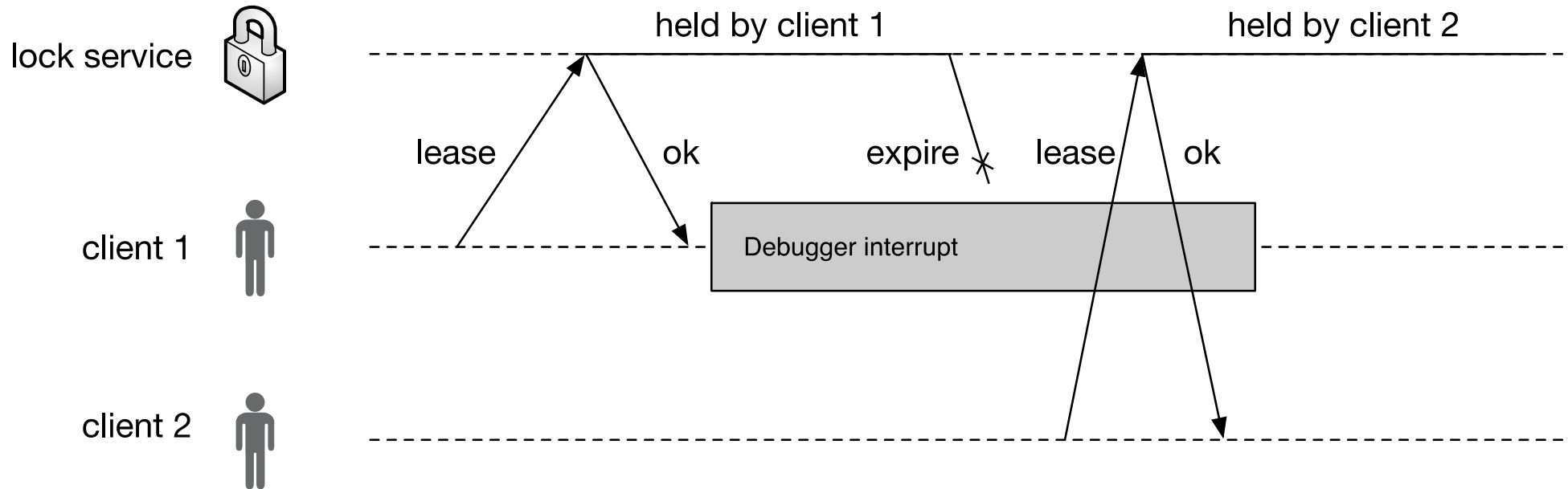
- ◆ **Programmierwerkzeuge** haben großen Einfluss
 - Komplexität der Fehlerbehandlung kritisch
 - Ungünstige Abstraktionen verwischen Fehlerursachen
- ◆ **OO: viele Abhängigkeiten** und Abstraktionsschichten
 - Proxy-Objekte und RMI „verbergen“ Verteilung
 - Interagierende Objekten oft nicht für Verteilung ausgelegt
- ◆ **Message Passing: Programmiermodell nah an der Realität**
 - Das Netzwerk operiert inhärent nachrichtenbasiert
 - Kein Bruch zwischen Anwendungs- und Systemsicht
 - Leichteres Auffinden von Ursache und Wirkung (Kausalbeziehungen der Nachrichten)

Zeitverhalten

- ◆ Verteilte Systeme sind **auf Timeouts angewiesen**
 - *Fehlererkennung*:
 - Langsame sind nicht von toten Knoten unterscheidbar
 - Ausfallerkennung nicht mit Sicherheit möglich (siehe versch. Failure Detector Strategien)
 - *Synchronisation*:
 - Lease Zeiten für geteilte Ressourcen (z.B. Distr. Lock)
 - Wettbewerbssituationen (z.B. Leader Election)

- ◆ Debugging mit **Breakpoints oft nicht praktikabel**
 - Andere Knoten laufen unverändert weiter
 - Untersuchte Knoten werden irrtümlich für tot erklärt
 - Eingriff in Systemverhalten / Kausalitätsketten durch Ausbremsen

Zeitverhalten in verteiltem Locking



- ◆ Lock Service unabhängig von Clients
- ◆ Debuggen von Client 1 während des Lease schwierig:
 - Ausbremsen von Client 1 führt zu Lock-Verfall
 - Andere Clients nicht kontrolliert durch Debugger

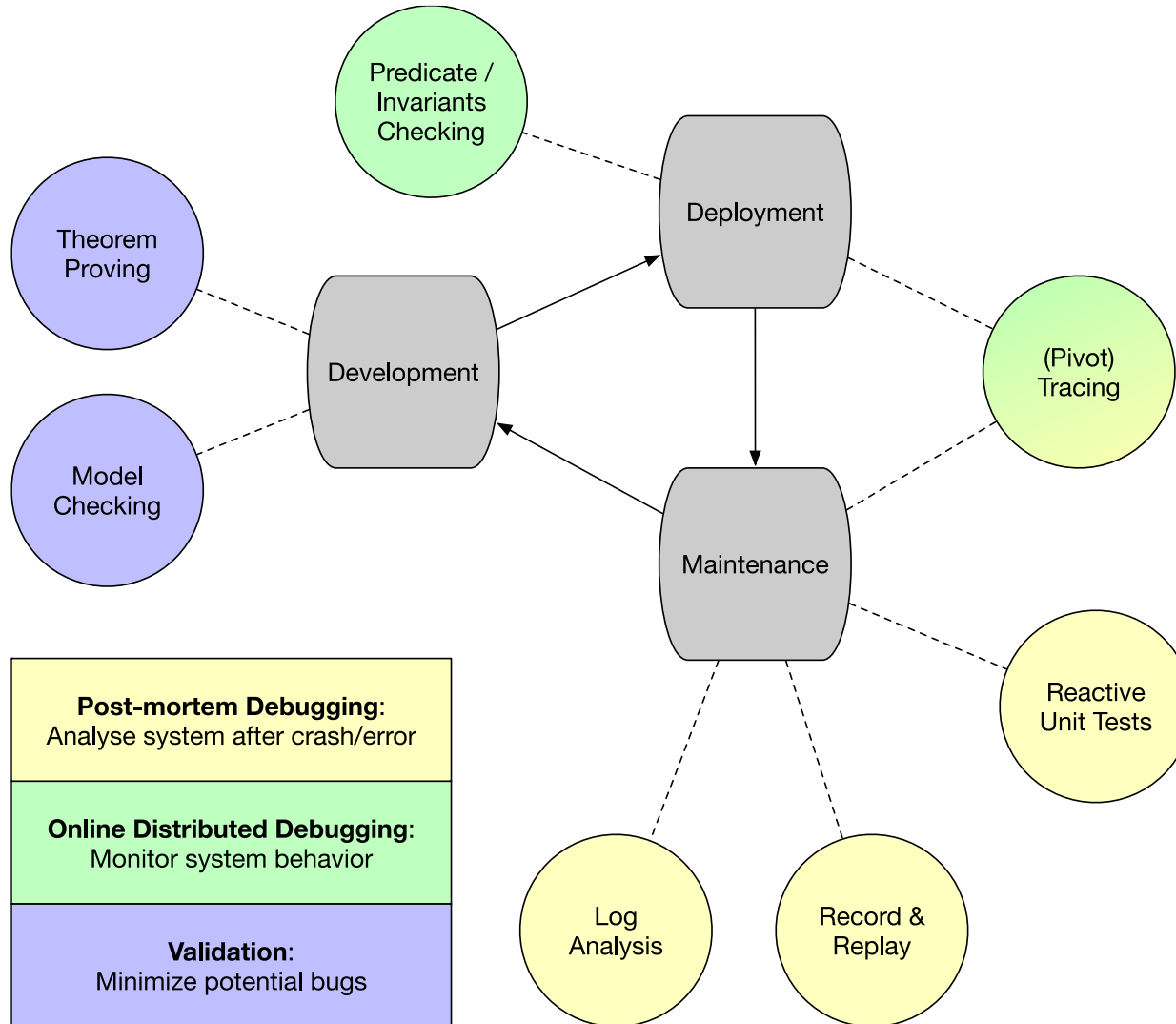
Lokalität & Nichtdeterminismus

- ◆ **Globaler Zustand** i.d.R. unbestimmbar
 - Zustand = Gesamtkonfiguration plus in-transit Nachrichten
 - Näherungsweise mit Snapshots „einfrierbar“
- ◆ **Keine gemeinsame Zeit**
 - Bestenfalls Happened-Before Beziehungen
 - Divergierende Sicht auf Reihenfolge von Fehlern
- ◆ **Reproduzieren von Fehlern** schwierig
 - Auffinden der Ereigniskette im verteilten System?
 - Einspeisen kritischer Ereignis-Folgen?

Methoden zur Fehlerfindung und – vermeidung in verteilter Software



Methodenübersicht



Methodenübersicht

- ◆ **Post-mortem Debugging:**
 - **Reaktive Unit Tests** – Reproduktion mit minimierten Tests
 - **Record and Replay** – Deterministische Reproduktion
 - **Tracing** – Auswerten aufgezeichneter Kommunikation
 - **Log-Analyse** – Auswerten von Debug-Nachrichten
- ◆ **Online Distributed Debugging:**
 - **Pivot Tracing** – Überwachung der Kommunikation zur Laufzeit durch Stichproben
 - **Predicate/Invariants Checking** – Erkennen kritischer Systemzustände während der Laufzeit
- ◆ **Validierung:**
 - **Model Checking** – automatisiertes, erschöpfendes Testen
 - **Theorem Proving** – Ausschluss von Fehlern in der Spezifikation einer Software

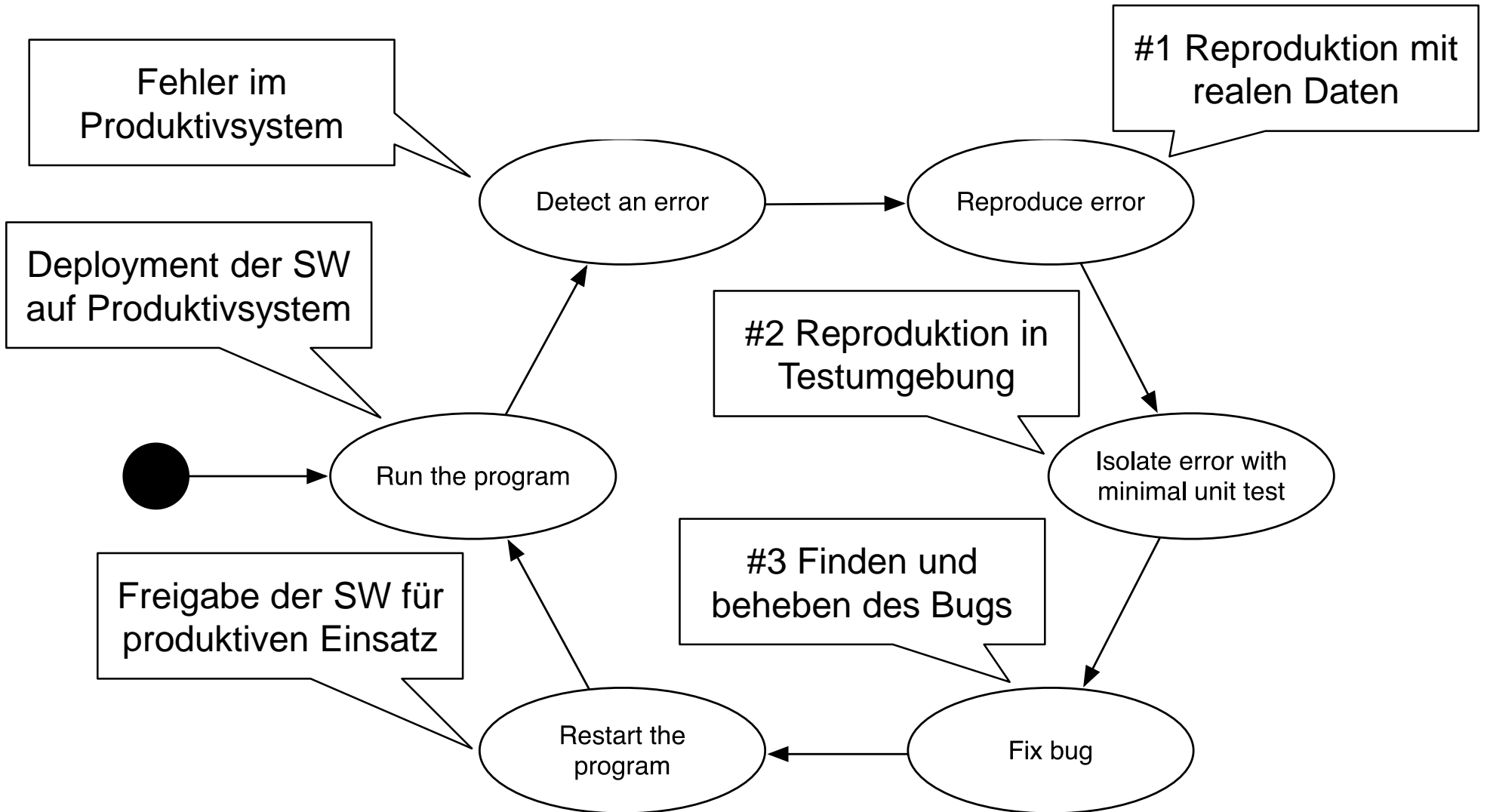
Post-mortem Debugging



Reaktives Unit Testing

- ◆ Reproduktion logischer Fehler in **minimaler Testumgebung**
- ◆ **Nicht** auf Artefakte (z.B. Logs) des Deployments angewiesen
- ◆ **Simulation** kritischer Nachrichten oder Ereignisketten
 - Z.B.: Fehlernachricht bei Handshake
 - Einfacher bei nachrichtenbasierter Programmierung
- ◆ **Vergleichsweise einfach**, jedoch hohe Detektionsrate *
- ◆ In der **Praxis oft unstrukturiert / ad hoc**

Typischer Kreislauf reaktiver Tests

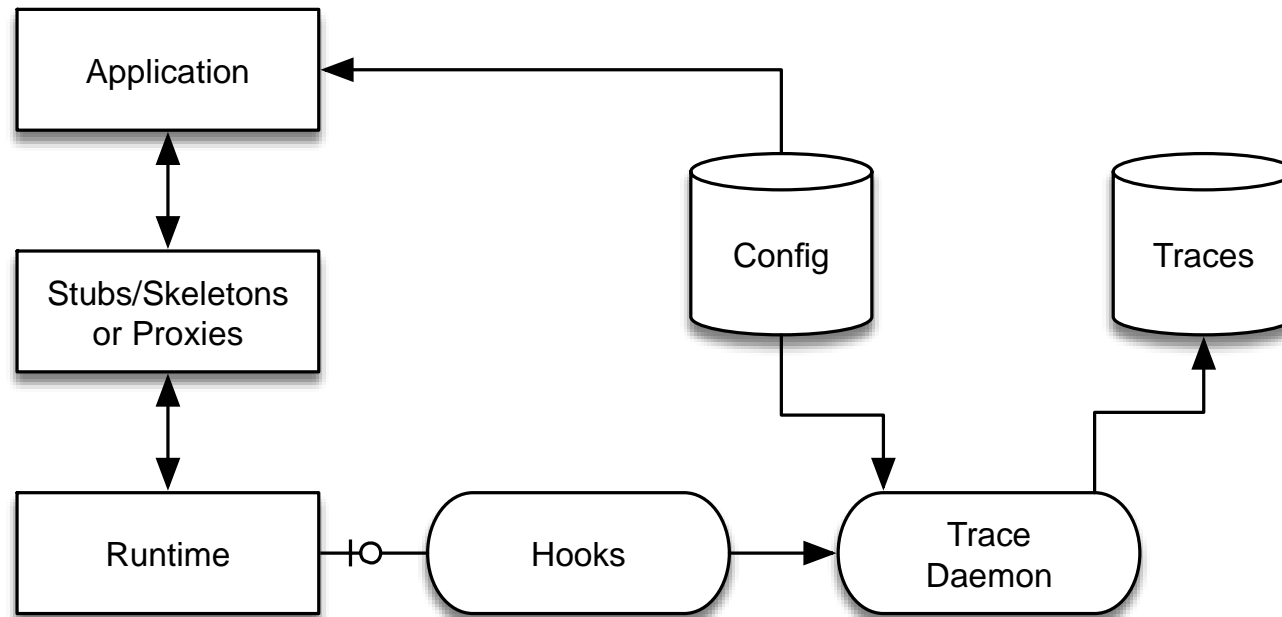


Record and Replay

- ◆ Record:
 - **Aufzeichnen** einer Programmausführung
 - **Einfangen** aller **nichtdeterministischen Ereignisse**
 - **Hoher Aufwand während** der **Laufzeit** des Programms
(mögliche Beeinflussung des Systemverhaltens durch verändertes Laufzeitverhalten)

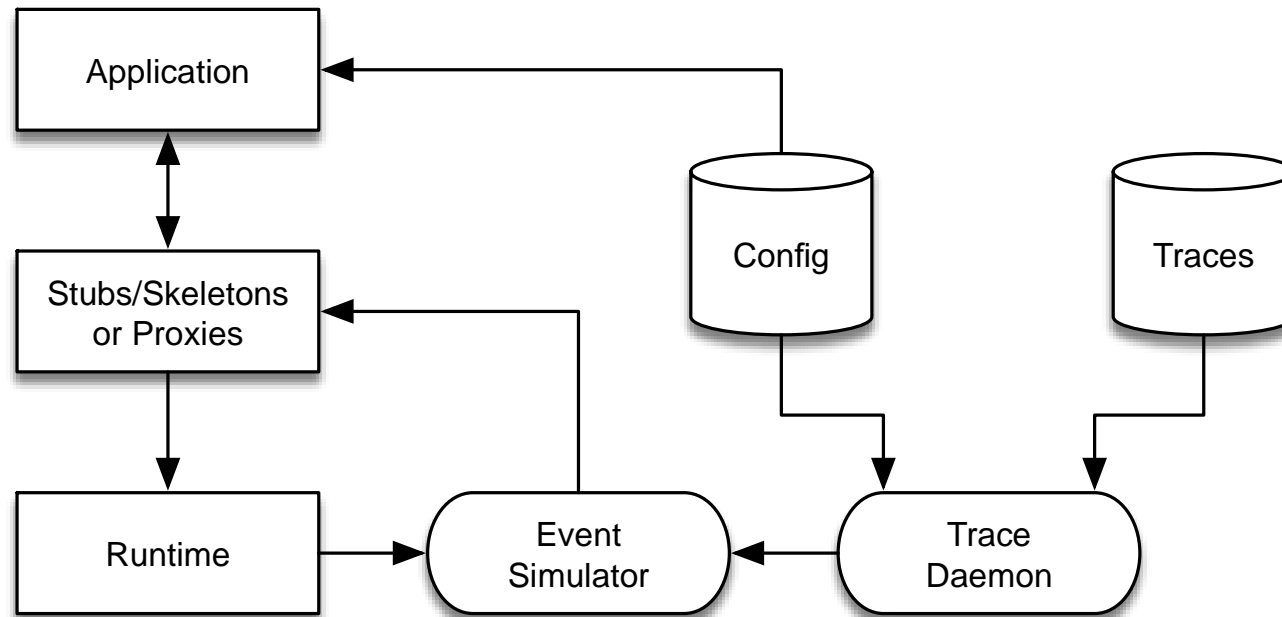
- ◆ Replay:
 - **Einlesen** des protokollierten Programmablaufes
 - **Wiedereinspielen** aufgezeichneter Ereignisse
 - Exakte **Schritt-für-Schritt Ausführung** der Aufzeichnung
 - **Debugger-Integration** möglich, z.B. in GDB *

Record Phase



- ◆ Anwendung kommuniziert über Proxy-Objekte zu Remotes
- ◆ Aufzeichnen aller Netzwerk- und I/O-Events über Hooks
- ◆ Speichern des Ablaufs als „Traces“ in Datei oder Datenbank

Replay Phase

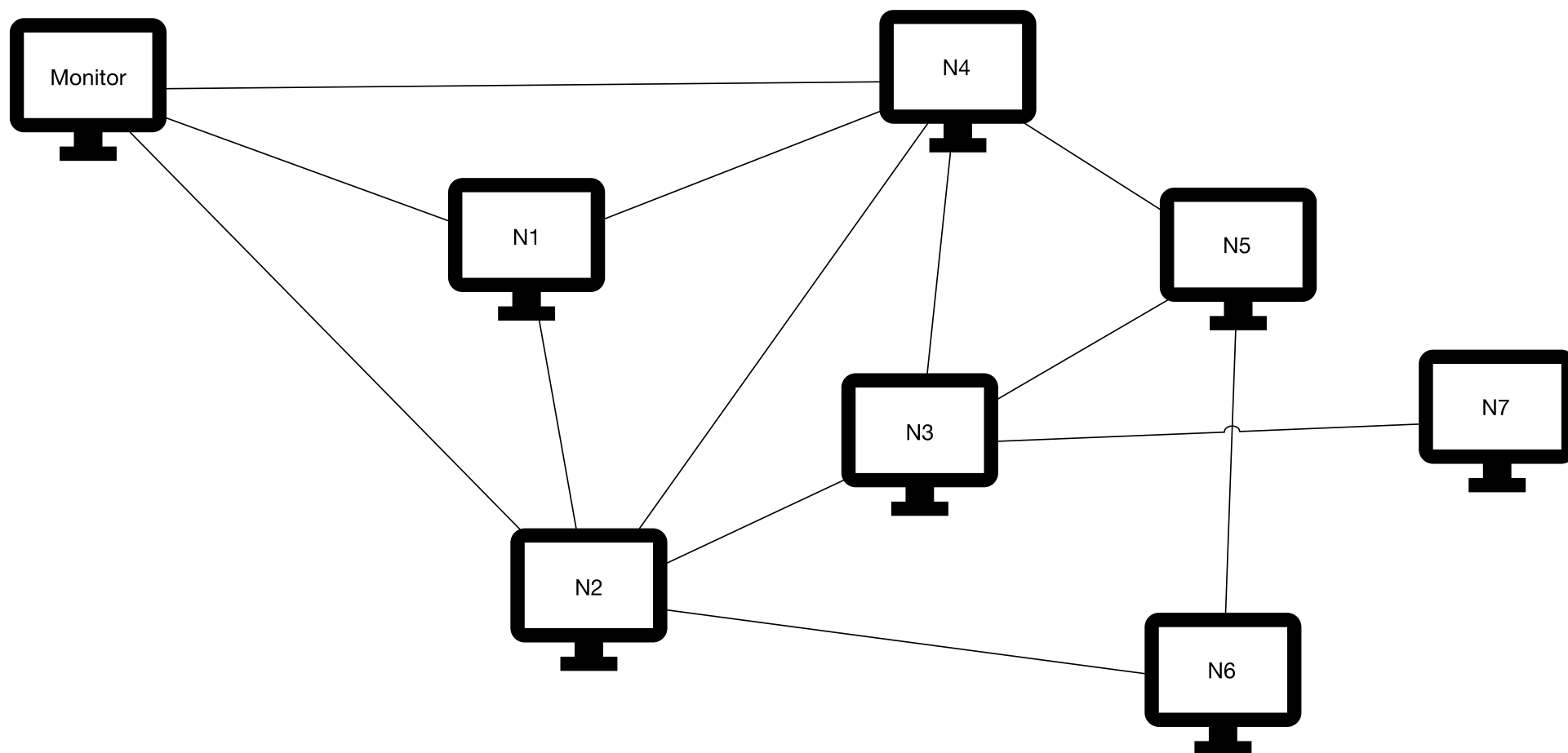


- ◆ Replay Phase ist transparent für lokale Anwendungsteile
- ◆ Wiedereinspielen aller Netzwerk- und I/O-Events aus Traces
- ◆ Deterministischer Programmablauf (in Debugger-Umgebung)

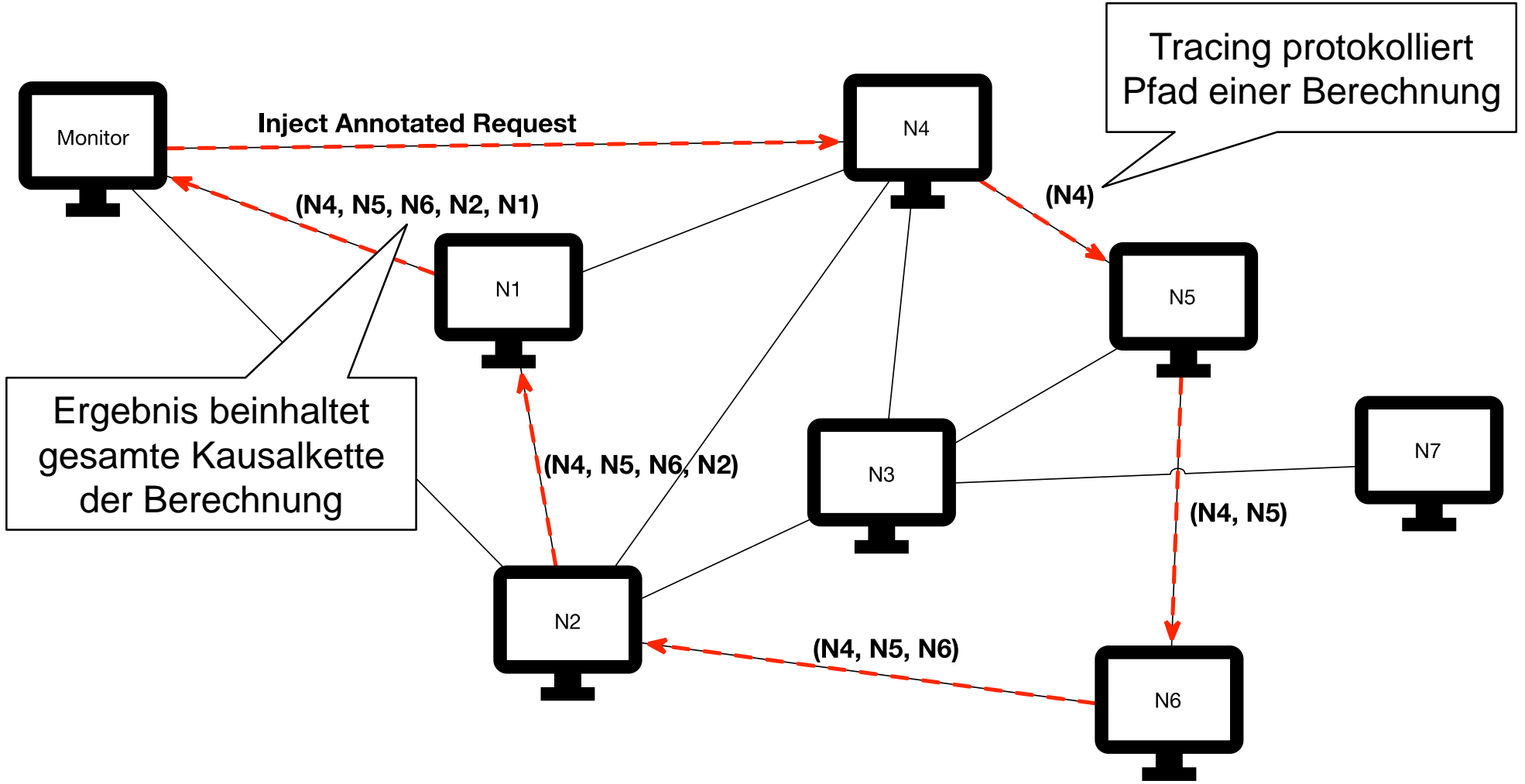
Tracing

- ◆ **Messen** von Datenflüssen
 - Annotation von Nachrichten mit **Metadaten**
 - Metadaten müssen in jedem Verarbeitungsschritt weitergereicht werden
- ◆ **Alle Teilsysteme** müssen am Tracing teilnehmen
 - Einfache Zuordnung von Inputs zu Outputs
 - Zeitliche und kausale Ordnung von Datenflüssen über Anwendungen, Protokolle, Datenbanken, etc. hinweg
- ◆ **Vollständiges Aufzeichnen** der Kommunikation und Metadaten
 - Reproduktion des Systemverhaltens und zeitlicher Abläufe
 - Auffinden logischer Fehler und Performanz-Probleme

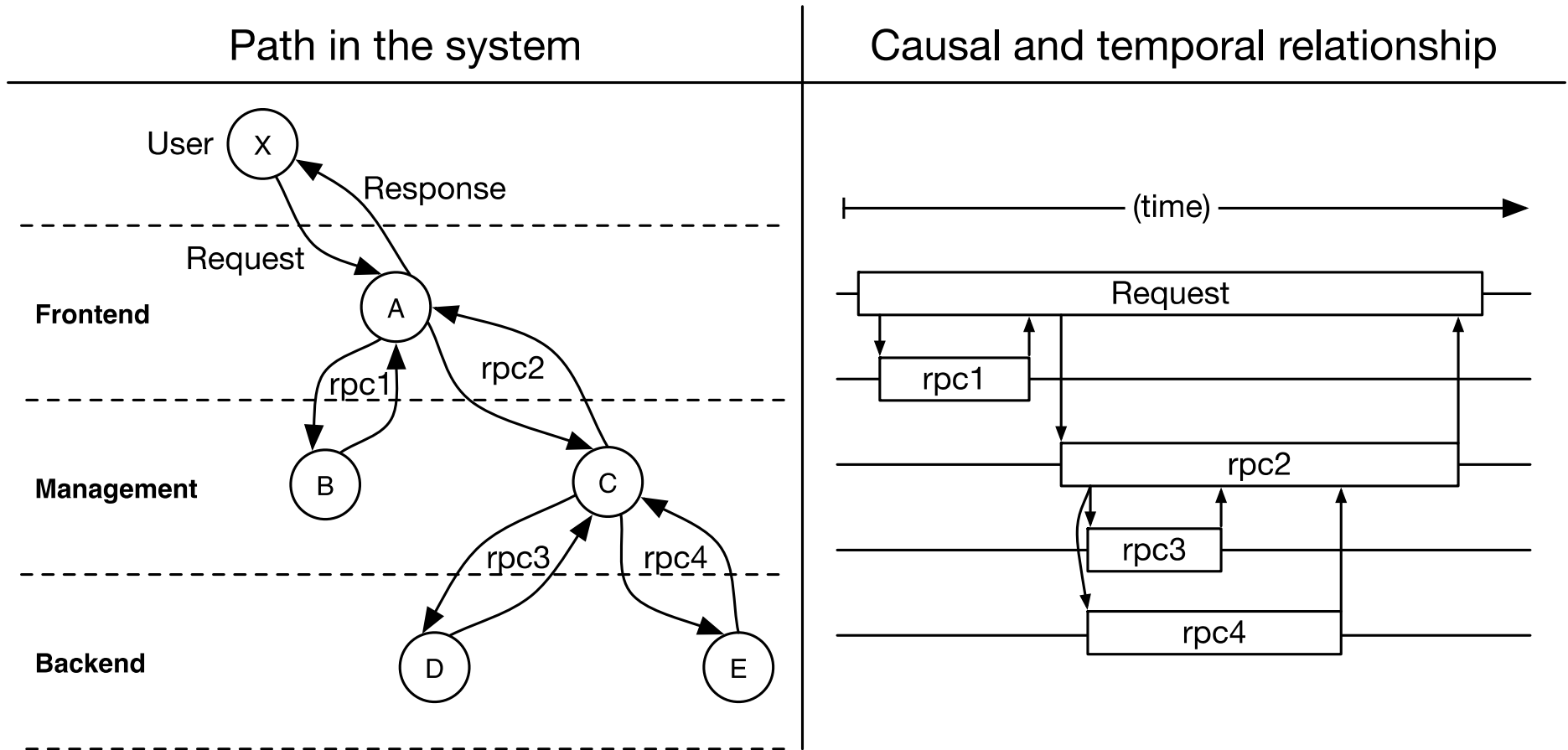
Tracing



Tracing

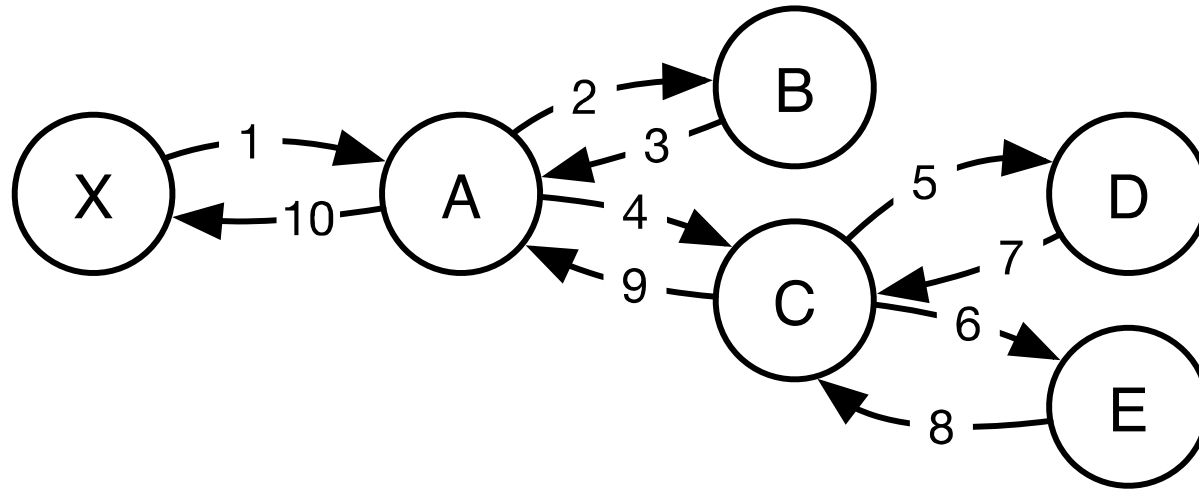


Kausale Ordnung durch Tracing



* Abb. modifiziert übernommen aus Benjamin Sigelman et al.

Kausale Ordnung durch Tracing

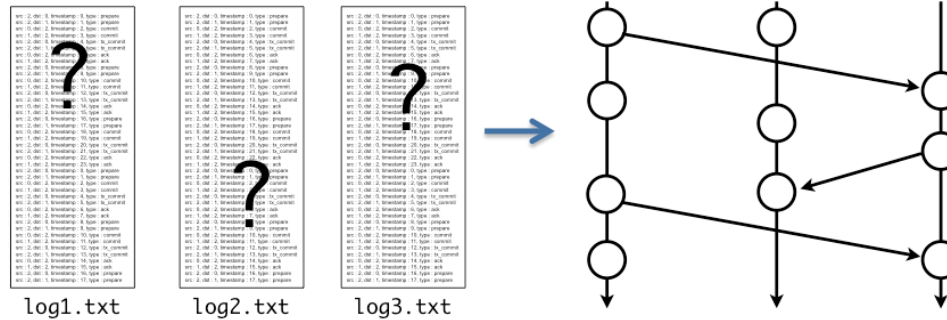


- ◆ Reproduktion der Nachrichtenpfade aus Metadaten
- ◆ Zuordnung von Inputs und Outputs erlaubt kausale Ordnung
- ◆ Timestamps erlauben Reproduktion zeitlicher Abläufe
- Aussagen über Abhängigkeiten und Berechnungsdauer

Log-Analyse

- ◆ **Auswerten von Konsolen-, Debug- oder Systemlogs**
 - i.d.R. bei beliebiger Software *ohne Änderung* möglich
 - **Leichtgewichtig**, aber oft zu detailliert ohne Tool-Support
- ◆ **Blackbox-Ansätze** (Auswertung ohne Quellcode-Zugriff):
 - Suche nach **charakteristischen Mustern** (manuell/Tool-gestützt oder automatisierte Anomalie-Erkennung mit Machine Learning *)
 - **Visualisierung** der Nachrichtenflüsse (z.B. mit ShiViz)
- ◆ **Whitebox-Ansätze** (Quellcode-Ebene):
 - Erfordert streng **strukturiertes Log-Format**
 - Visualisierung aufgezeichneter **Kontroll- und Nachrichtenflüsse** (z.B. mit Causeway)

ShiViz



- ◆ Visualisierung von Log-Dateien als interaktive Kommunikationsgraphen mit kausaler Ordnung
- ◆ Import beliebiger Log-Formate (Regex-basierter Importer)
- ◆ Anforderung: JSON-formatierte Vector-Timestamps
- ◆ Volltextsuche über Log-Ereignisse sowie strukturierte Suche nach Kommunikationsmustern (z.B. Request/Response oder Broadcast)
- ◆ Visuelle Diffs zum Vergleich mehrerer Programmdurchläufe

Volltext- und
Struktursuche

Entitäten im
System

ShiViz GUI

Search the visualization

SEARCH

INIT ; NAME = spawn_s
erver ; LAZY = true ; HID
DEN = true

timestamp: 14797300326
component: caf
level: DEBUG
host: spawn_server
class: caf.scheduled
function: launch
file: scheduled ac

5 collapsed events

SPAWN ; ID = 1 ; ARGS = (actor config)

SPAWN ; ID = 2 ; ARGS = (actor config)
/ INIT ; NAME = spawn_server ; LAZY = tr
ue ; HIDDEN = true

INIT ; NAME = config server ; LAZY = tru
7 collapsed events
14 collapsed events

SPAWN ; ID = 3 ; ARGS = (actor config)
14 collapsed events

SPAWN ; ID = 4 ; ARGS = (actor config)
INIT ; NAME = timer actor ; LAZY = false

INIT ; NAME = printer actor ; LAZY = fal
3 collapsed events
3 collapsed events

SPAWN ; ID = 5 ; ARGS = (actor config)
3 collapsed events

SPAWN ; ID = 6 ; ARGS = (actor config)
INIT ; NAME = scheduled actor ; LAZY = f

INIT ; NAME = scoped actor ; LAZY = fals

Happened-
Before
Beziehung

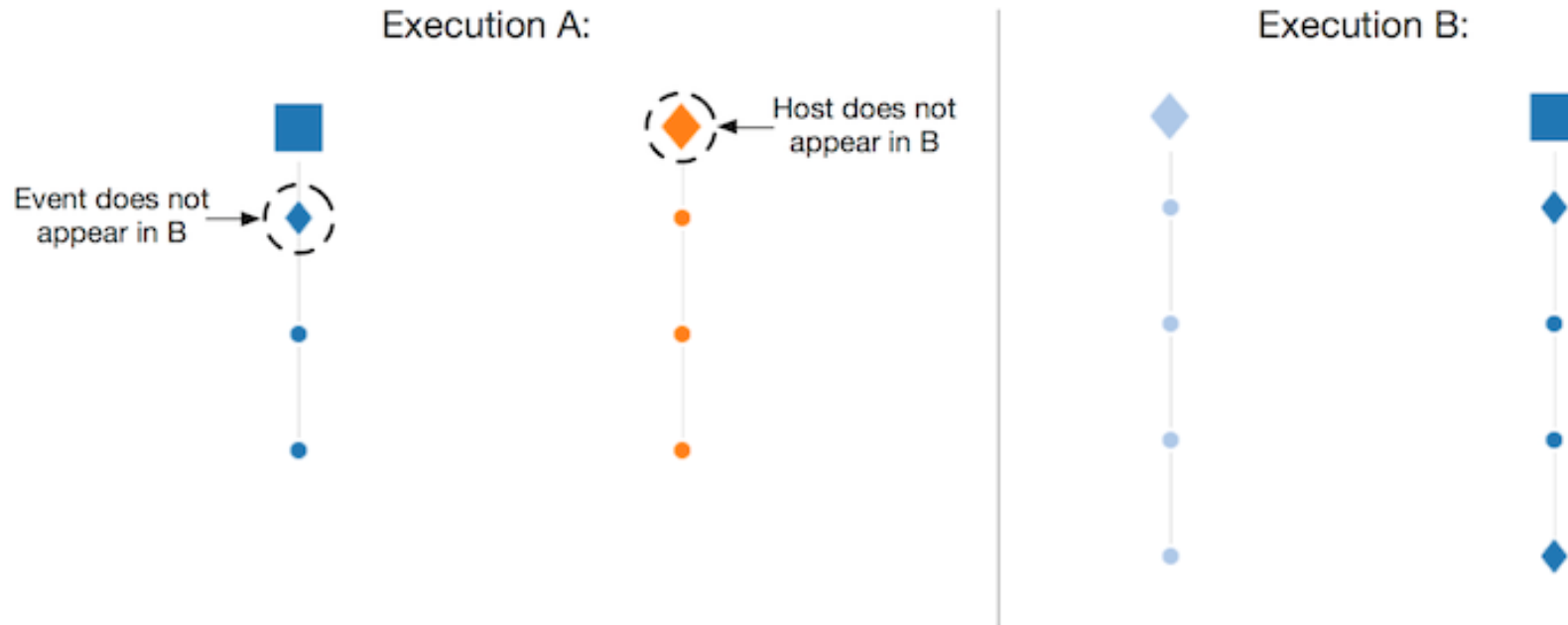
Jeder Kreis ist
ein Event

INIT ; NAME = spawn_
server ; LAZY = true ;
HIDDEN = true

timestamp: 1479730032
613682000
component: caf
level: DEBUG
host: spawn_serve
r1
class: caf.schedule
d_actor
function: launch
file: scheduled_a
ctor.cpp
line: 166

Einträge in der
visualisierten
Log-Datei

ShiViz Visual Diff



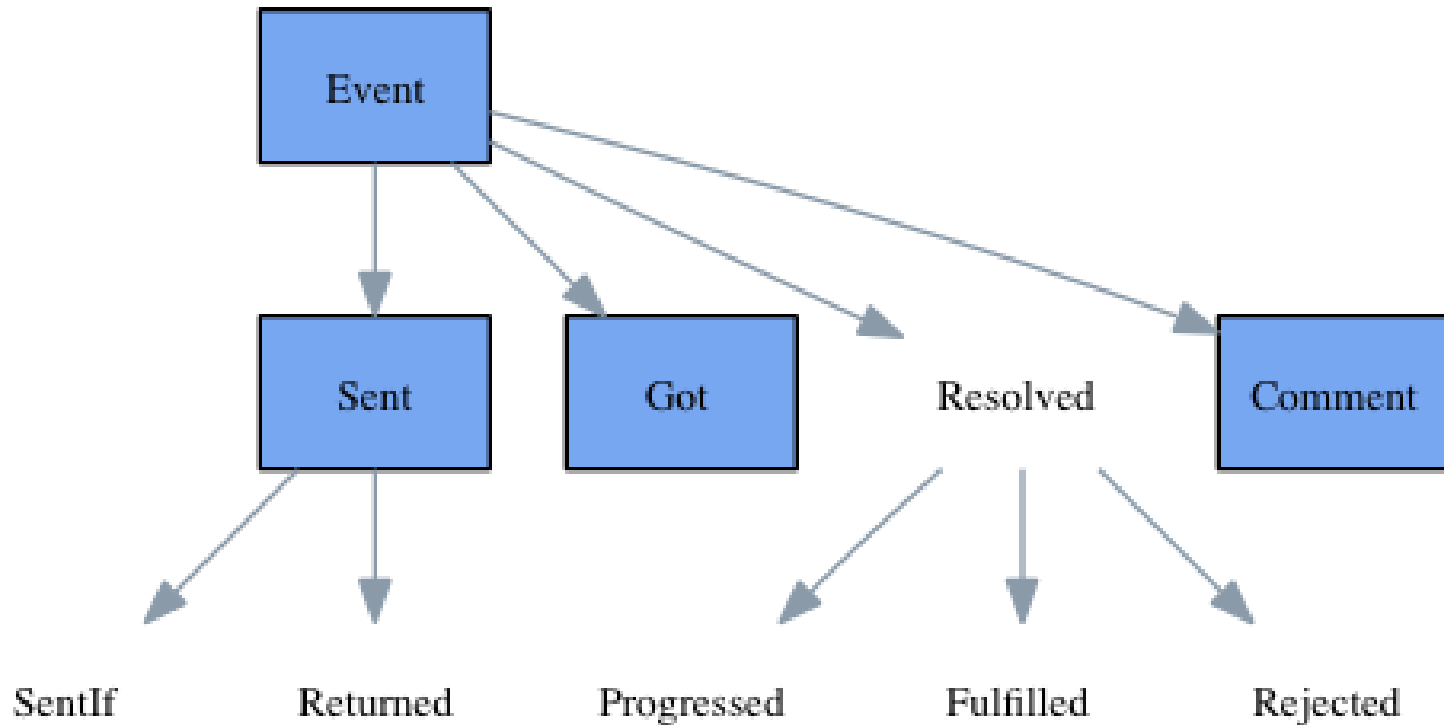
- ◆ Per-Event vergleich zweier Programmläufe
- ◆ Events/Hosts die nur in einem der verglichenen Läufe vorkommen sind symbolisiert mit ◆
- ◆ Erlaubt schnelles auffinden divergierender Abläufe

Causeway

- ◆ Nachrichtenbasierter „**Distributed Debugger**“ zum Verständnis von Programmverhalten und Korrektheit
- ◆ Kontrollfluss muss sich mit **Nachrichten und Promises** beschreiben lassen
- ◆ JSON-basiertes **Log-Format** mit festen Event-Kategorien:
 - Sent: Versand einer Nachricht
 - Got: Empfang einer Nachricht
 - Comment: Zusätzliche, optionale Kontext-Informationen
 - Resolved: State-Änderungen eines Promise

Causeway Events

Trace Log Event Types



Causeway GUI

Entitäten im System

Nachrichten-Flüsse im System

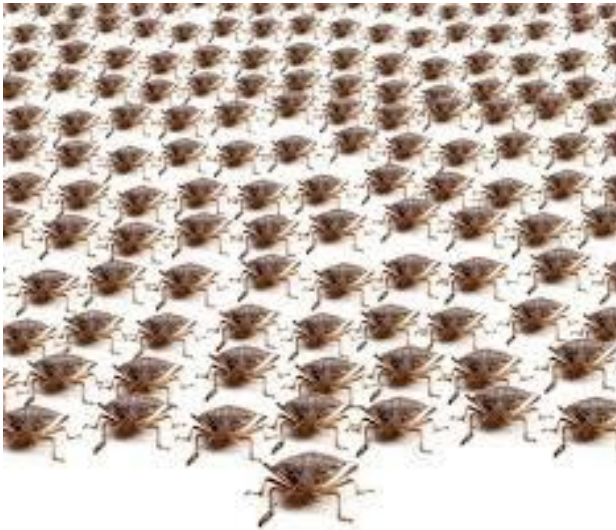
Aktueller Call-Stack zur angezeigten Quellcode-Stelle

Quellcode-Stelle an der die aktuelle Nachricht gesendet oder verarbeitet wird

The screenshot displays the Causeway GUI interface, which is divided into several panes:

- Message Order Tree (Top Right):** A tree view showing the sequence of messages and actions. The root is '[buyer, 3]', which branches into several asynchronous actions like 'partInStock', 'checkCredit', and 'canDeliver'. Each action is further detailed with sub-messages and responses.
- Stack Explorer (Bottom Left):** A call stack view showing the current execution context. The top entry is '[buyer, 12] AsyncAnd.DoAnswer.fulfill', followed by '[buyer, 11] AsyncAnd.run', and then various 'tell' and 'canDeliver' messages.
- Source Code Editor (Bottom Right):** A code editor showing the implementation of the 'fulfill' method in 'org/waterken/purchase_ajax/AsyncAnd.java'. The current line is highlighted, showing the 'tellAreAllTrue' call.
- Entity Explorer (Top Left):** A pane showing system entities and commands, such as 'searchStacks' and 'findMultiples'.

Online Distributed Debugging



Pivot Tracing

- ◆ **Echtzeit-Monitoring** durch Sampling von Stichproben *
 - Leichtgewichtiges, aktives Messen
 - Überwachung der Antwortzeiten und Nachrichtenpfade
- ◆ **Dynamische Anpassung** des Tracing
 - Einspeisung neuer Messungen zur Laufzeit
 - Korrelation verschiedener Events
- ◆ Auch relevant als **Administrationswerkzeug**
 - Troubleshooting im laufenden Betrieb
 - Auffinden von Fehlkonfiguration und langsamer Knoten

Predicate/Invariants Checking

- ◆ **Deklarative Definition** von Programm-Invarianten
 - Beschreiben valider Systemzustände nach dem Muster „wenn A gilt, dann muss auch B gelten“
 - Festlegen von Abhängigkeiten und Gültigkeitsräumen verarbeiteter Daten
- ◆ **Kontinuierliche Überprüfung** während der Laufzeit
 - Vor und nach dem Verarbeiten von Daten
 - Bei Zustandsübergängen eines Teilsystems
- ◆ **Fehlerbehandlung** bei Verletzung deklarerter Invarianten
 - Fallback: „Selbstheilung“ durch festgelegte Übergänge in sichere Zustände
 - Debugging: Anhalten der Software zur Inspektion oder Abbruch mit aufgezeichneter Fehlerursache

D³S

- ◆ **DSL** zur Formulierung von *globalen* Prädikatsfunktionen (z.B. „keine zwei Maschinen dürfen denselben Lock exklusiv halten“)
- ◆ **Echtzeit-Überprüfung** von Snapshots des Systems
- ◆ Erlaubt das **Einfügen von Prädikaten zur Laufzeit**
- ◆ **Typische Prädikate** ~100-200 Zeilen lang mit maximalem Laufzeit-Overhead ~8% *
- ◆ Microsoft-spezifische Lösung (**nicht** Open Source)

D³S Beispielprädikat

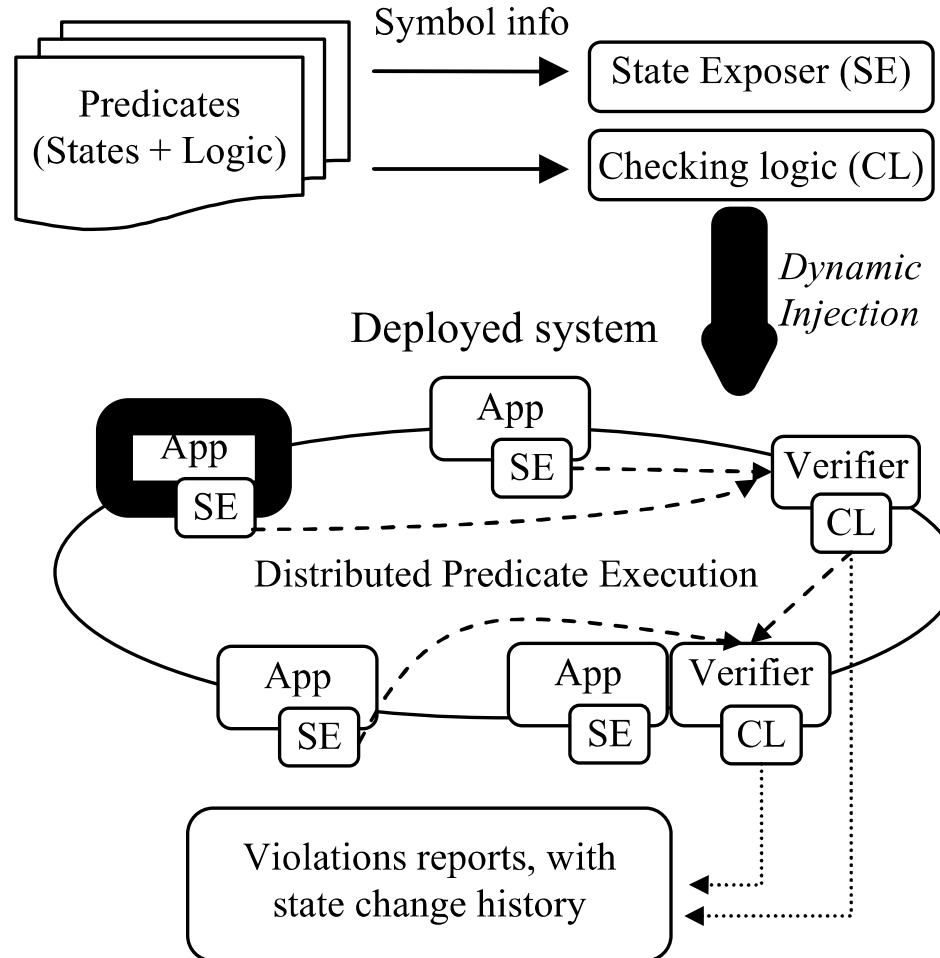
```
V0: exposer    → { ( client: ClientID, lock: LockID, mode: LockMode ) }  
V1: V0        → { ( conflict: LockID ) } as final  
after (ClientNode::OnLockAcquired) addtuple ($0->m_NodeID, $1, $2)  
after (ClientNode::OnLockReleased) deltuple ($0->m_NodeID, $1, $2)
```

Part 1: define the dataflow and types of states, and how states are retrieved

```
class MyChecker : vertex<V1> {  
    virtual void Execute( const V0::Snapshot & snapshot ) {  
        .... // Invariant logic, writing in sequential style  
    }  
    static int64 Mapping( const V0::tuple & t ) ; // guidance for partitioning  
};
```

Part 2: define the logic and mapping function in each stage for predicates

D³S Architektur



Validierung

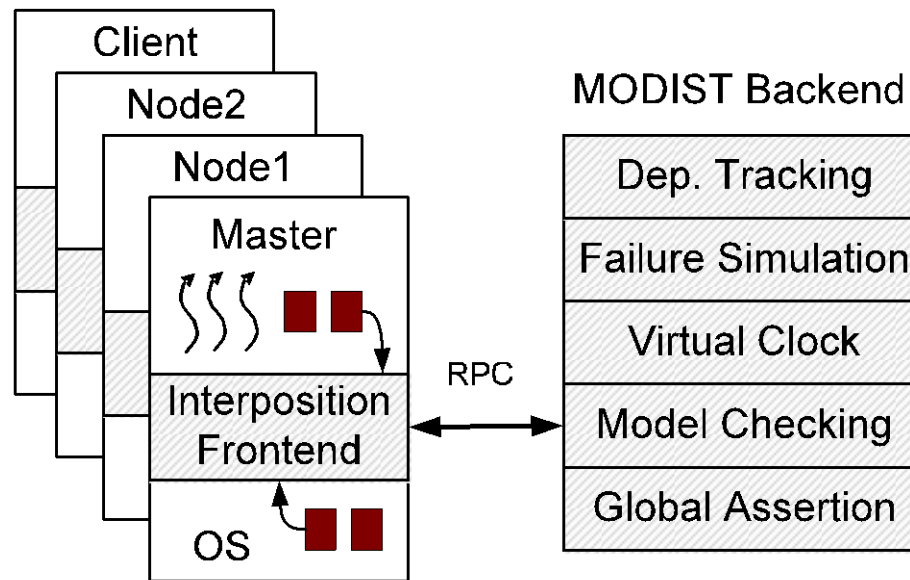


Model Checking

- ◆ **Erschöpfendes**, automatisches **Testen** eines Programms
 - **Maschinenlesbare Definition** gültiger Systemzustände, aus denen Testfälle abgeleitet werden
 - **Großer Zustandsraum** limitiert Skalierbarkeit in der Praxis
- ◆ **Symbolisch:**
 - Mathematische Modellierung des gesamten Systems als Zustandsautomat inklusive der Kommunikationskanäle
 - Ausführungen „symbolisch“ anhand des Modells
- ◆ **Explicit-State:**
 - Kontrolliertes Ausführen des tatsächlichen Programms
 - I.d.R. nur bis zu vordefinierter Ausführungstiefe

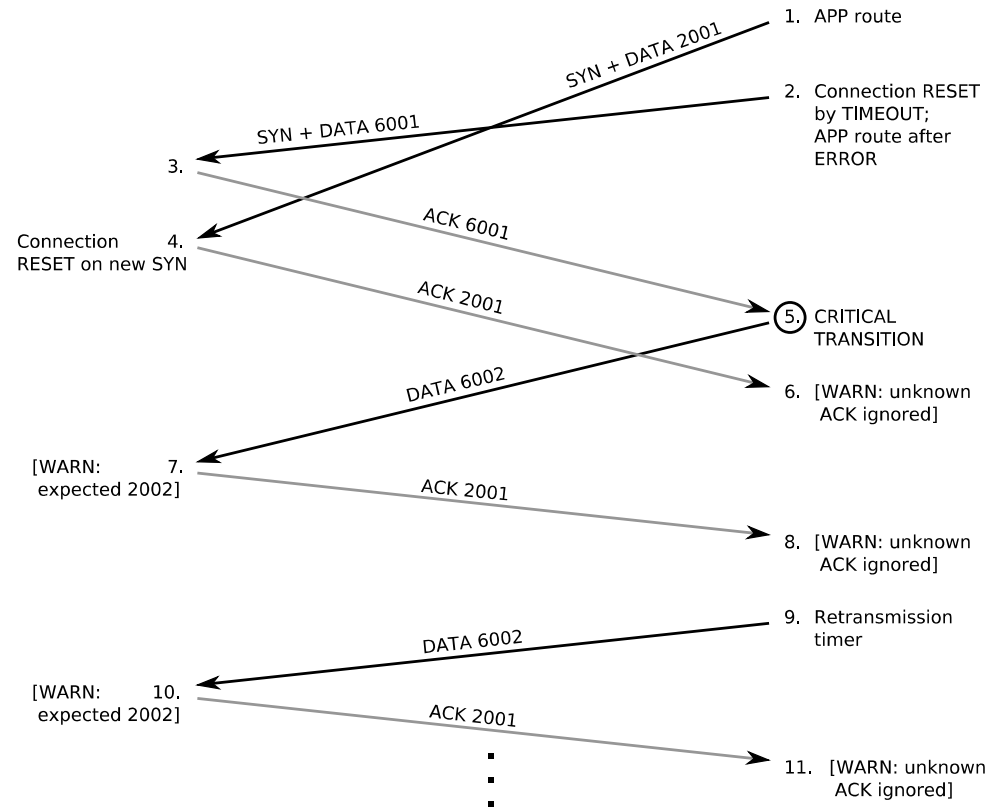
MoDist

- ◆ Blackbox Ansatz
- ◆ Transparentes Model Checking von unveränderten Systemen
- ◆ Ausführ-Engine zwischen Betriebssystem und Anwendung
- ◆ Simulationsumgebung zur deterministischen Ausführung verteilter Anwendungen mit virtueller Uhr



MaceMC

- ◆ Whitebox Ansatz
- ◆ Benutzerdefinierte Treiber-Software zur Initialisierung des Systems, Generierung von Input-Events und Überwachung von Systemeigenschaften
- ◆ Aufspüren kritischer Transitionen durch automatisches Testen und Generierung von Event-Graphen nach Auffinden kritischer Systemzustände



Theorem Proving

- ◆ **Maschinenlesbare**, mathematische **Spezifikation** der Eigenschaften eines Systems
 - **Modellierung** von Zustandsübergängen
 - Maschinengestütztes **Beweisen** von gewünschtem Systemverhalten unter allen Bedingungen
- ◆ **Problem**: Implementierung muss dem Modell entsprechen
 - Generierung oder Verifikation der Implementierung
 - Nur mit Werkzeugunterstützung praktikabel
- ◆ Hoher Aufwand, **spezialisierte Tools**, und **Expertenwissen** erforderlich

TLA+

- ◆ Spezifikationssprache für verteilte Anwendungen (aufgeteilt in Module)
- ◆ Beschreibung von States, Verhalten, Invarianten, Transitionen, etc.
- ◆ Operationen und Datenstrukturen auf Basis von Mengentheorie und Logik
- ◆ Findet Widersprüche in der Spezifikation, bzw. in Modulen
- ◆ Entwickelte Spezifikationen können zum automatisierten Testen (Model Checking) einer bestehenden Implementierung benutzt werden

TLA+ IDE

The screenshot displays the TLA+ IDE interface. The main editor window shows a TLA+ module named 'Firewall.tla' with the following code:

```
1 ----- MODULE Firewall -----
2 EXTENDS Integers
3 CONSTANTS Address, /* The set of all addresses
4 Port, /* The set of all ports
5 Protocol /* The set of all protocols
6
7 AddressRange == /* The set of all address ranges
8 Address \X Address
9
10 InAddressRange[r \in AddressRange, a \in Address] ==
11 /\ r[1] <= a
12 /\ a <= r[2]
13
14 PortRange == /* The set of all port ranges
15 Port \X Port
16
17 InPortRange[r \in PortRange, p \in Port] ==
18 /\ r[1] <= p
19 /\ p <= r[2]
20
21 Packet == /* The set of all packets
22 [sourceAddress : Address,
23 sourcePort : Port,
24 destAddress : Address,
25 destPort : Ports,
26 protocol : Protocol]
27
28 Firewall == /* The set of all firewalls
29 [Packet -> BOOLEAN]
30
31 Rule == /* The set of all firewall rules
32 [remoteAddress : AddressRange,
33 remotePort : PortRange,
34 localAddress : AddressRange,
35 localPort : PortRange,
36 protocol : SUBSET Protocol,
37 allow : BOOLEAN]
38
39 Ruleset == /* The set of all firewall rulesets
40 SUBSET Rule
41
42 Allowed[rset \in Ruleset, p \in Packet] == /* Whether the ruleset allows the packet
43 LET matches == {rule \in rset :
44 /\ InAddressRange[rule.remoteAddress, p.sourceAddress]
45 /\ InPortRange[rule.remotePort, p.sourcePort]
46 /\ InAddressRange[rule.localAddress, p.destAddress]
47 /\ InPortRange[rule.localPort, p.destPort]
48 /\ p.protocol \in rule.protocol}
49 IN /\ matches /= {}
50 /\ \A rule \in matches : rule.allow
51 -----
```

The error message in the 'Parsing Errors' window is:

```
TLA+ Parser Error
line 25, col 16 to line 25, col 20 of module Firewall
Unknown operator: 'Ports'.
```

The status bar at the bottom indicates '25 : 21', '179M of 329M', and 'Spec Status: error'.

Praktische Herangehensweise



Neue Software

- ◆ **Nachrichtenbasierte Programmierung**
(aktive Objekte oder Aktoren)
 - Kleine, leicht zu testende Komponenten
 - Keine Seiteneffekte durch geteilten Speicher
- ◆ **Hochstehende Middleware**
 - Abstraktion von Byte-basierten Primitiven (z.B. Sockets)
 - Kausale Zuordnung von Input/Output Nachrichten
 - Testmodus zur deterministischen Simulation (*Mocking*) von Netzwerk-Events und verschiedenen Topologien
- ◆ **Testgetriebene Entwicklung**
 - Unit Tests für einzelne Komponenten
 - Integrationstests für Zusammenspiel von Komponenten
 - Bei kritischer Software: Model Checking

Bestehende Software

- ◆ **Migration** hin zu nachrichtenbasierter Programmierung und hochstehender Middleware (MW)
 - **Kapseln** bestehender Komponenten
 - Identifikation unabhängiger Programmteile
 - Isolation durch **nachrichtenbasierte Fassaden**
 - **Ausweiten** vorhandener Tests
 - Anbindung an Netzwerk-Simulationsmodus der MW
 - Testen der gekapselten Komponenten und deren Zusammenspiel
 - **Erweitern** des Systems
 - Festlegung der Schnittstellen zu Legacy Komponenten
 - Neuer Code nur noch auf Basis nachrichtenbasierter Programmierung

Werkzeugeinsatz

- ◆ **Visualisieren** von verteilten Systemen erlaubt schnelleres Verständnis von komplexen Zusammenhängen
 - Auf **bestehende Software** anwendbar
 - Evtl. **Anpassung des Log-Formates** (z.B. für Causeway)
- ◆ Das Zusammenspiel **vielschichtiger (Web) Services** lässt sich **mit Tracing analysieren**
 - I.d.R. auf **bestehende Software** anwendbar (z.B. Dapper), aber Anpassung der Software erlaubt besseren Einblick in Systemverhalten durch vollständigere Trace-Informationen
 - **Pivot Tracing** erlaubt leichtgewichtiges Überwachen einer Anwendung im Produktivbetrieb

Werkzeugeinsatz

- ◆ Record & Replay erlaubt **exakte Wiedergabe eines einzelnen Knoten** im Netzwerk
 - Dedizierte Werkzeuge und hoher Laufzeit-Overhead
 - Löst das Problem von Nichtdeterminismus in einer Debugger-Umgebung
- ◆ **Formale Methoden** (Model Checking, Theorem Proving)
 - Bei **Neuentwicklung** kritischer Systemteile
 - **Spezifizieren** (z.B. mit TLA+) und automatisches Testen
- ◆ Predicate/Invariants Checking erlaubt **exakte Reproduktion eines fehlerhaften globalen Zustandes**
 - **Ergänzung** bestehender Software, z.B. durch Integration von D³S
 - Besonders von Interesse bei Software die vom Entwicklerteam auch betrieben wird (z.B. **Microservices**)

Weiterführende Literatur

- ◆ Ding Yuan et al., **Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems**, in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.
- ◆ Wei Xu et al., **Experience Mining Google's Production Console Logs**, in *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, 2010.
- ◆ Jonathan Mace et al., **Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems**, in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.
- ◆ Benjamin Sigelman et al., **Dapper, a Large-Scale Distributed Systems Tracing Infrastructure**, *Google Technical Report*, 2010.
- ◆ Ivan Beschastnikh et al., **Debugging Distributed Systems: Challenges and Options for Validation and Debugging**, in *ACM Queue Volume 14 Issue 2*, 2016.

- ◆ Xuezheng Li et al., **D3S: Debugging Deployed Distributed Systems**, in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.
- ◆ Chris Newcombe et al., **How Amazon Web Services Uses Formal Methods**, in *Communications of the ACM Volume 58 Issue 4*, 2015.
- ◆ Junfeng Yang et al., **MODIST: Transparent Model Checking of Unmodified Distributed Systems**, in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.
- ◆ Charles Killian et al., **Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code**, in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, 2007.