# Verteilte Systeme

## Verteiltes Debugging

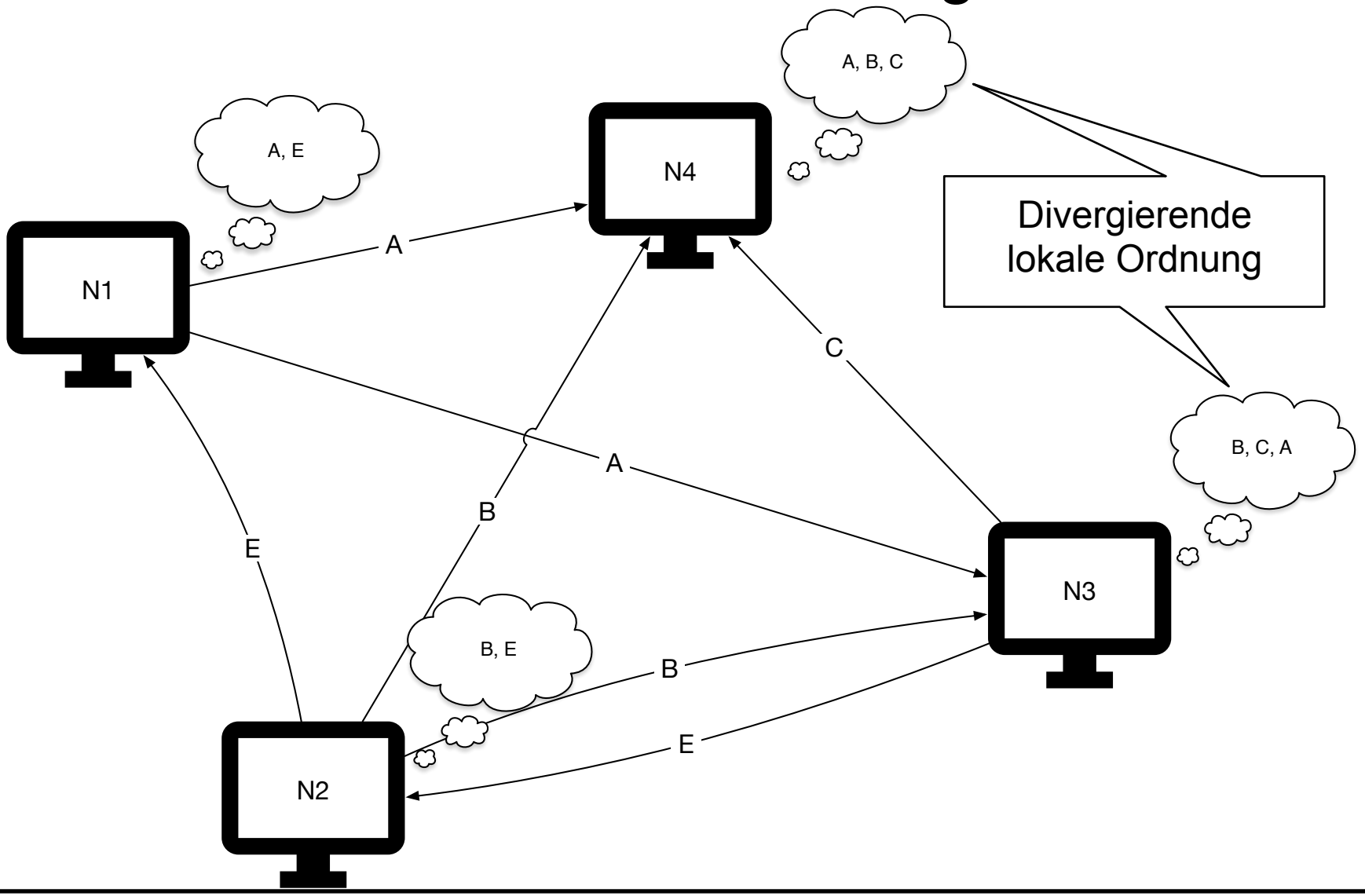
# Debugging

- 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
    - **Lokation** 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 gesamten State einer laufenden Anwendung
  - RAM: Speicherverbrauch und Inhalt Heap-allokierter Daten
  - Register und Stack: Variablen innerhalb der aktuellen Funktion und aller aufrufenden Funktionen
- **Modifizieren** von State und Code
  - Überschreiben von Speicherinhalten
  - Quellcode-Änderung in laufenden Programmen (*just in time debugging*)

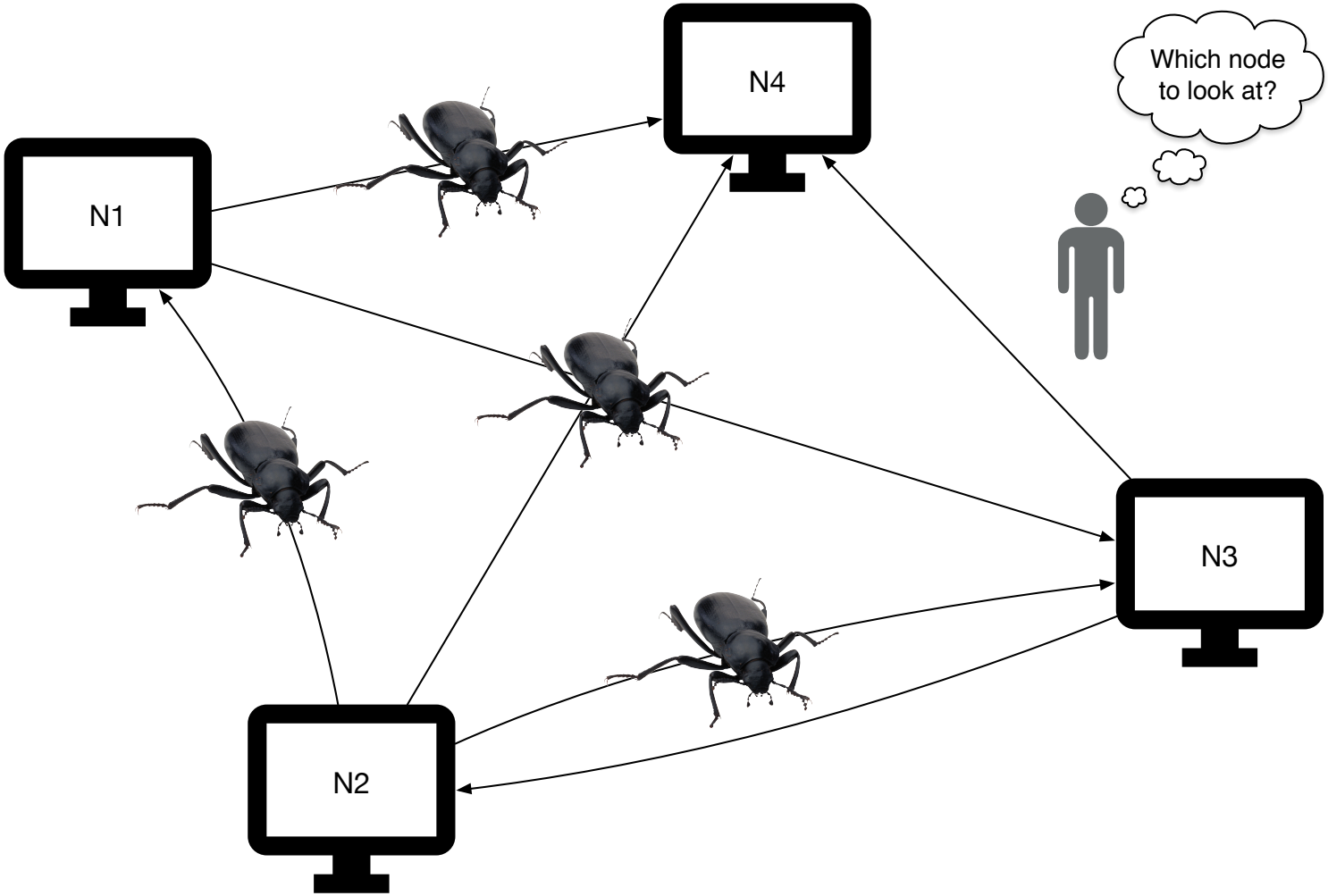
# Verteilte Ausführung



# Verteilte Ausführung

- **Lokale Ordnung**:
  - Kein Knoten sieht alle Ereignisse
  - Latenz und Topologie entscheiden Ereignisreihenfolge
  - Zeitliche Ordnung impliziert *nicht* Kausalität
- **Nichtdeterminismus**:
  - Scheduler entscheidet Verarbeitungsort und -reihenfolge
  - Unterschiedliche Laufzeiten gleicher Aufgaben
  - Mehrfache Programmdurchläufe können unterschiedliche Ereignisketten produzieren

# Debugging verteilter Anwendungen



# 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/schwer nachstellbar in Testaufbauten

# Verteilte Systeme und Fehler

- **Ausfälle** sind *häufig* (Beispielrechnung):
  - Ein Server fällt im Schnitt alle 10.000 Stunden aus
  - Bei 10.000 Servern fällt im Schnitt pro Stunde einer aus
- Die **Fehlerrate** *steigt* je mehr Knoten ein System hat
  - Die MTBF (MeanTimeBetweenFailures) sinkt proportional
  - Mehr Fehlerquellen wie z.B. Festplatten und Kabel
- **Aber**: Robustheit *wächst ebenfalls* mit der Anzahl der Knoten
  - Ein Komplettausfall wird unwahrscheinlicher
  - Mehr Knoten bedeuten mehr Redundanz

# Besonderheiten verteilter Software

- **Fehlerwahrscheinlichkeit proportional zu 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 trotz Teilausfällen sein

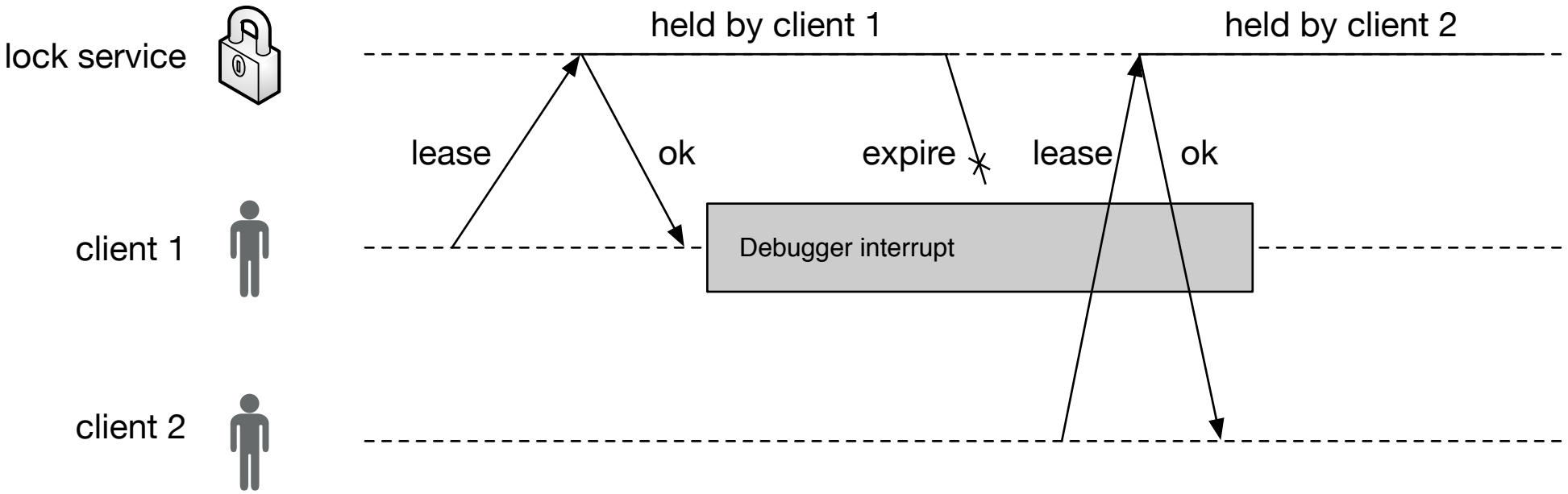
# Grenzen von Debuggern

- **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:
  - *Timing*: Timeouts und variierende Kausalitätsketten
  - *Lokalität*: unabhängige Speicherbereiche / Variablen
  - *Nichtdeterminismus*: globaler Zustand nicht bestimmbar

# Timing

- Verteilte Systeme sind **auf Timeouts angewiesen**
  - *Fehlererkennung*:
    - Langsame sind nicht von toten Knoten unterscheidbar
    - Ausfallerkennung nicht mit 100% 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

# Timing in Distributed Locking




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

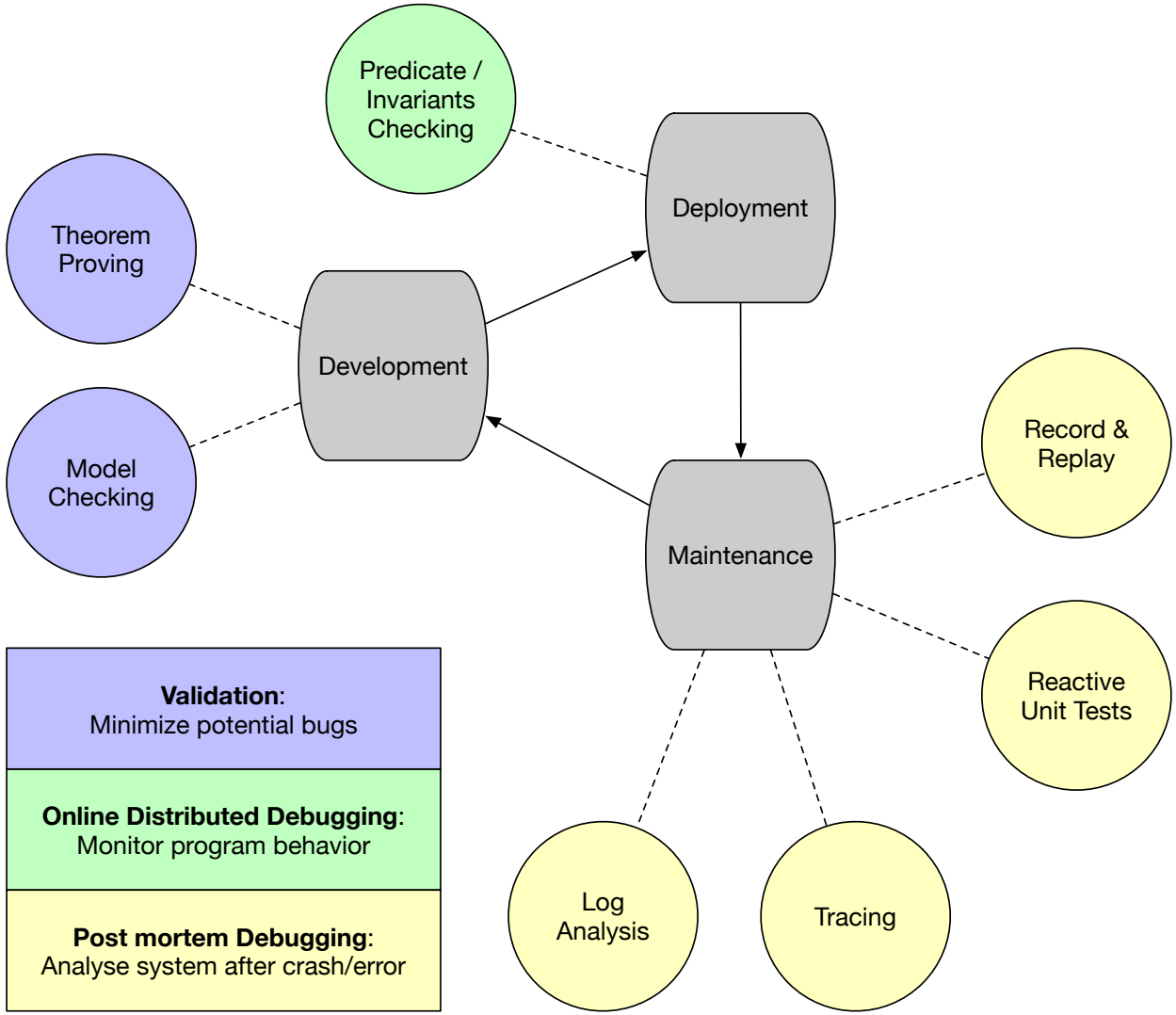
# Lokalität & Nichtdeterminismus

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

# Wahl 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

# Debugging & Entwicklungsphasen



# Verteiltes Debugging: Übersicht

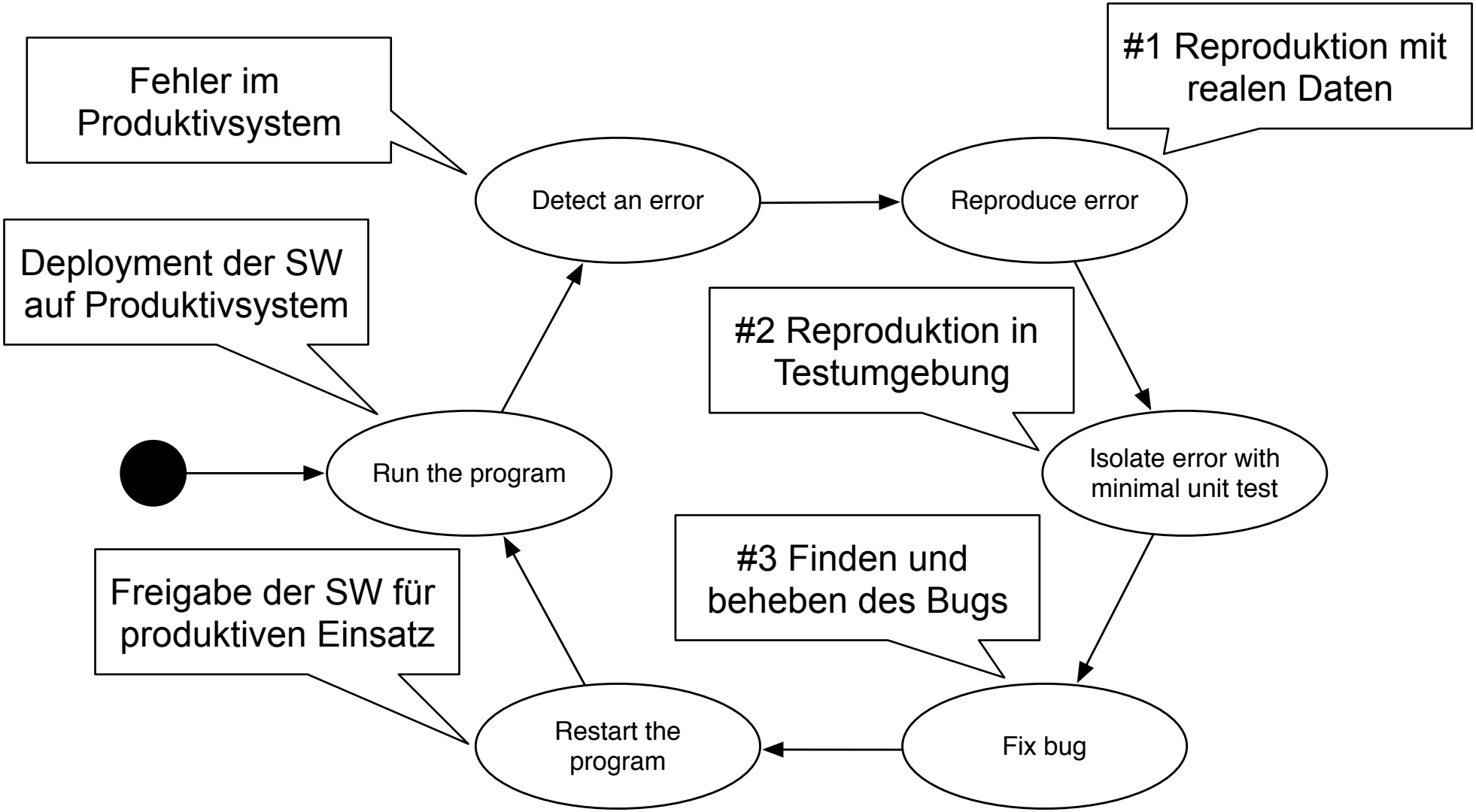
- **Post mortem Debugging**:
  - *Reaktive Unit Tests* (Reproduktion mit minimierten Tests)
  - *Log-Analyse* (Auswerten von Debug-Nachrichten)
  - *Tracing* (Auswerten aufgezeichneter Kommunikation)
  - *Record and Replay* (Deterministische Reproduktion)
- **Online Debugging**:
  - *Predicate/Invariants Checking* (Erkennen krit. Zustände)
- **Validierung als Alternative zu Debugging**:
  - *Model Checking* (Erschöpfendes Testen)
  - *Theorem Proving* (Ausschluss von Fehlern in Spezif.)

# Reaktives Unit Testing

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

* Weitere Informationen in Ding Yuan et al.

# Typischer Kreislauf reaktiver Tests

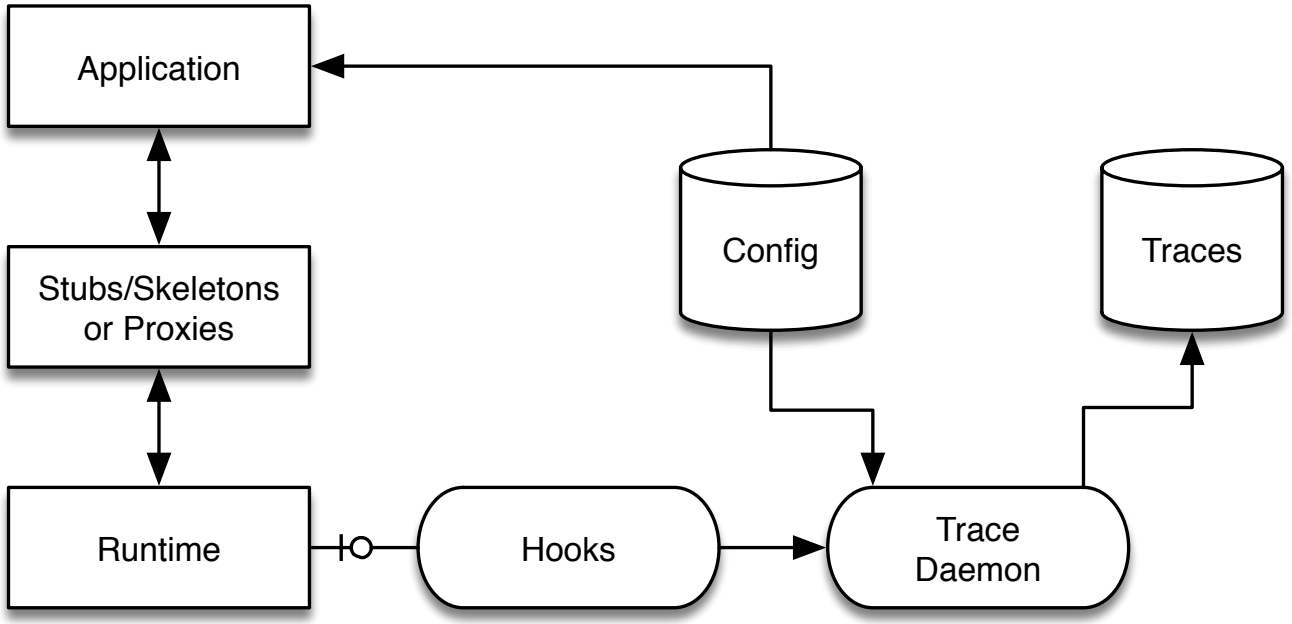


# Record and Replay

- Record:
  - **Aufzeichnen** einer Programmausführung
  - **Einfangen** aller **nichtdeterministischer Events**
  - **Hoher Aufwand während** der **Laufzeit** des Programms (mögliche Beeinflussung des Systemverhaltens durch veränderte Timings)
- Replay:
  - **Einlesen** des protokollierten Programmablaufes
  - **Wiedereinspielen** aufgezeichneter Events
  - Exakte **Schritt-für-Schritt Ausführung** der Aufzeichnung
  - **Debugger-Integration** möglich, z.B. in GDB *

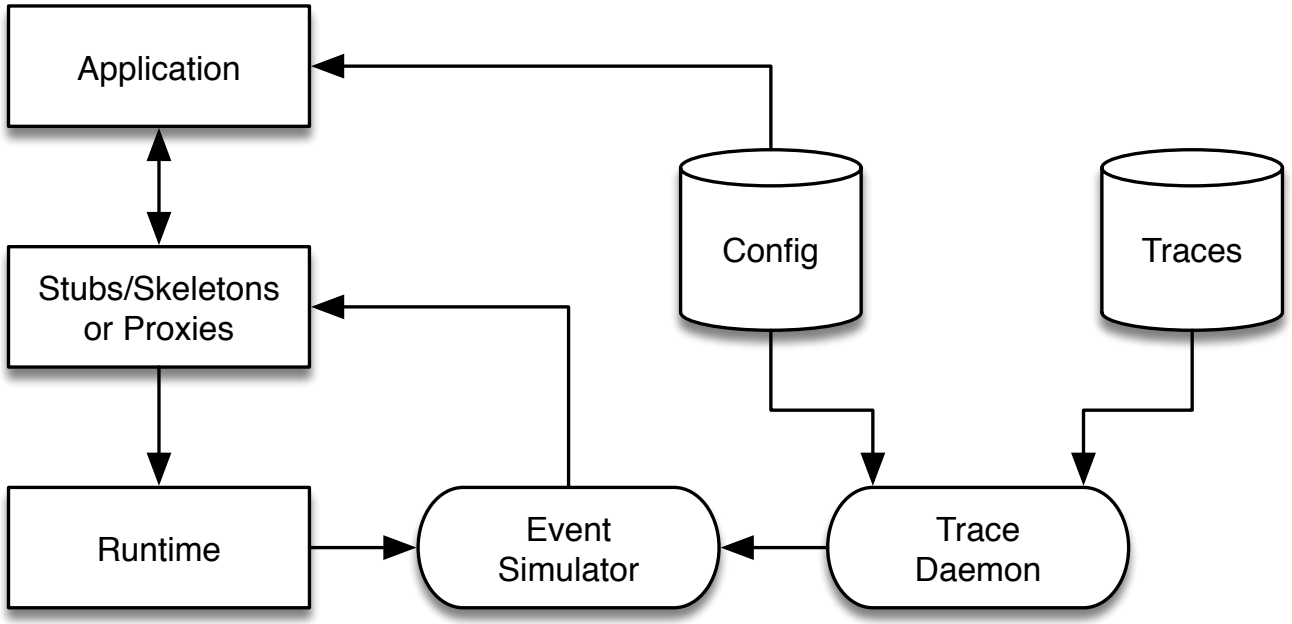
* Weitere Informationen in Dennis Geels et al.

# Record Phase




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

# Replay Phase



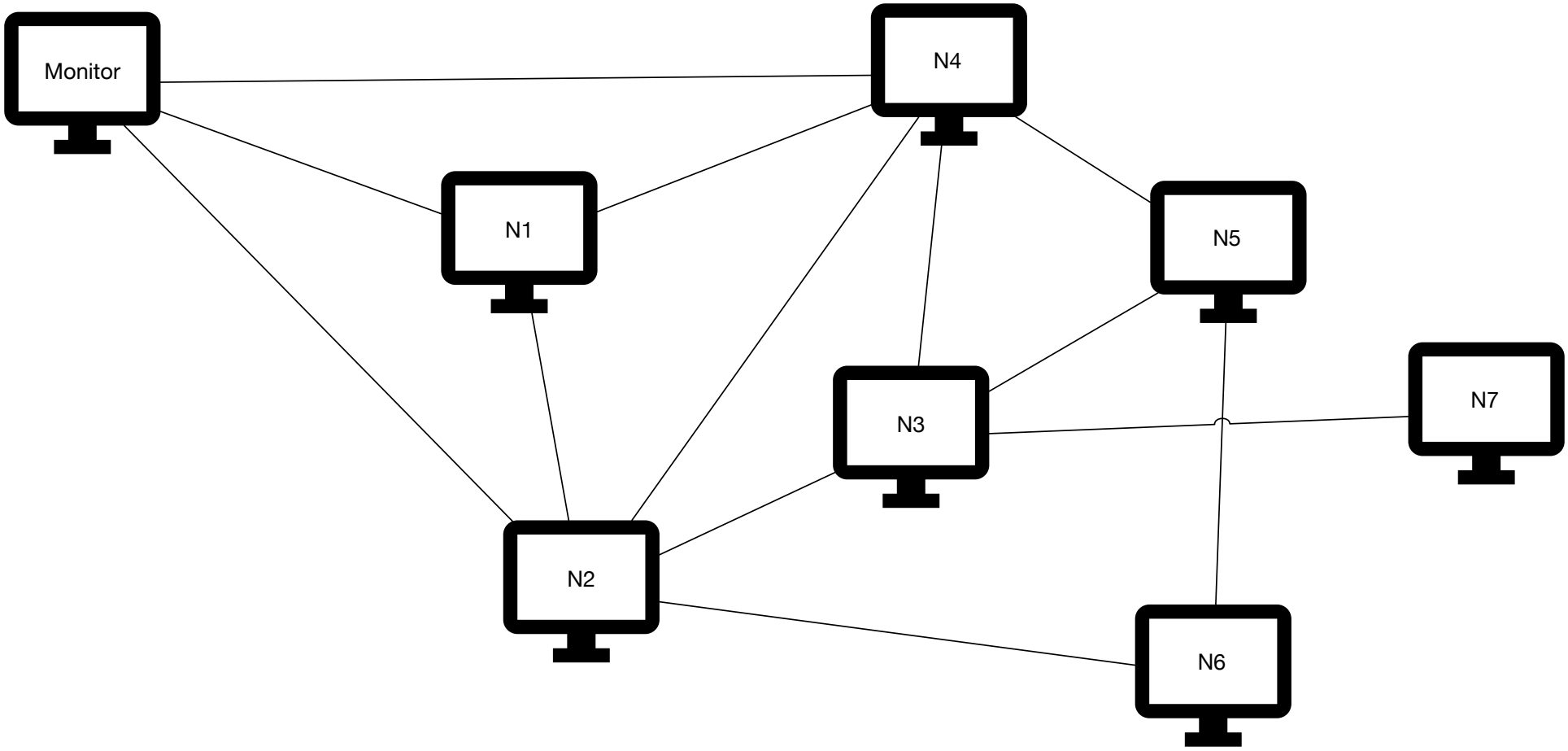

- Replay Phase ist transparent für lokale Anwendungsteile
- Wiedereinspielen aller Netzwerk- und I/O-Events aus Traces
- Deterministischer Programmablauf (in Debugger-Umgebung)

# Tracing

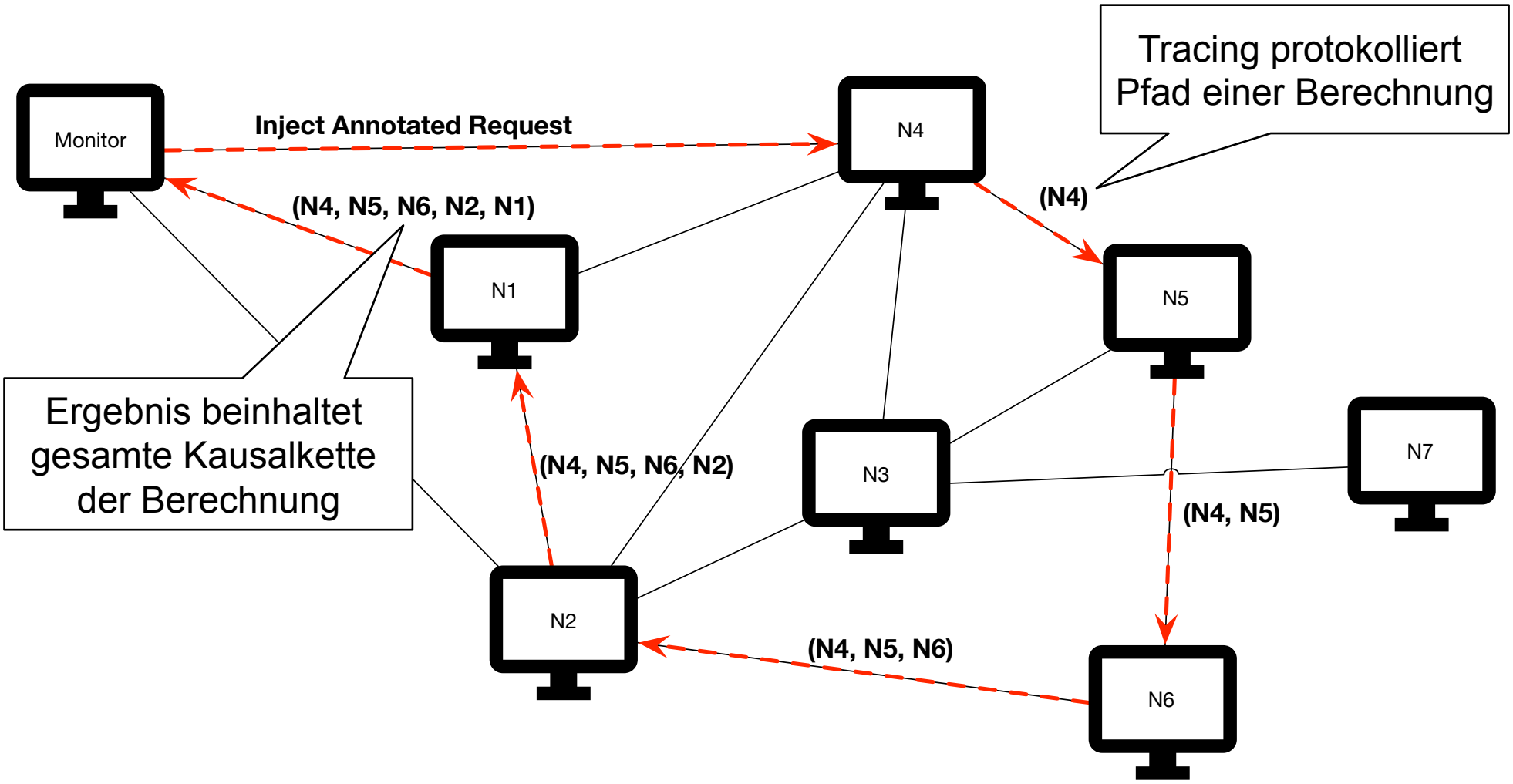
- **Leichtgewichtiges Messen** von Datenflüssen
  - Annotation von Nachrichten mit Metadaten
  - Metadaten müssen in jedem Verarbeitungsschritt weitergereicht werden
- **Alle Teilsysteme** müssen am Tracing teilnehmen
  - Einfache Zuordnung von Inputs zu Outputs
  - Zeitliche und Kausale Ordnung von Datenflüssen über Anwendungen, Protokolle, Datenbanken, etc. hinweg
- Vollständiges **Aufzeichnen oder Stichproben**
  - Pivot Tracing: Echtzeit-Monitoring durch Sampling *
  - Vollstände Traces: Reproduktion von Systemverhalten und Messen von Performance (z.B. Dapper)

* Weitere Informationen in Jonathan Mace et al.

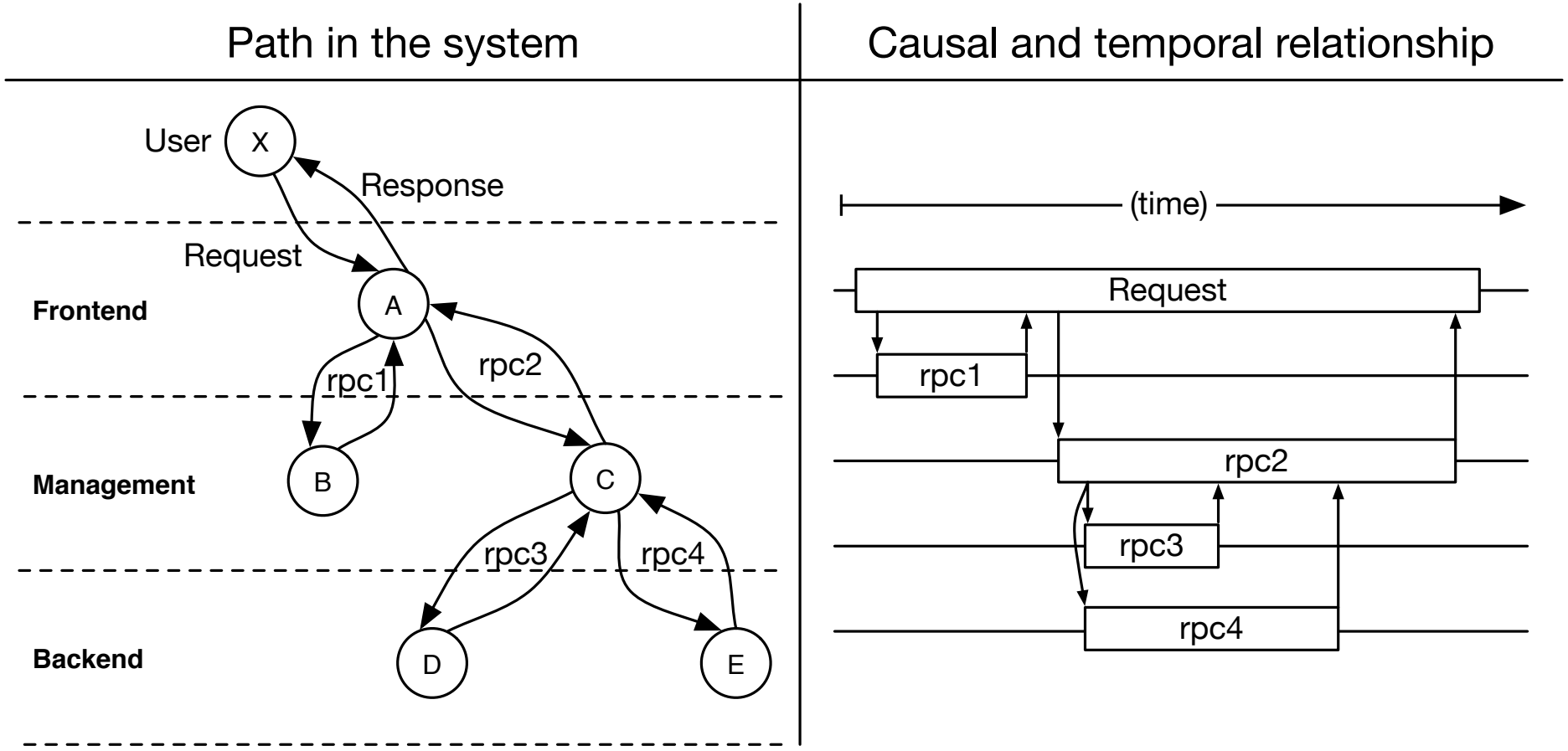
# Tracing



# Tracing

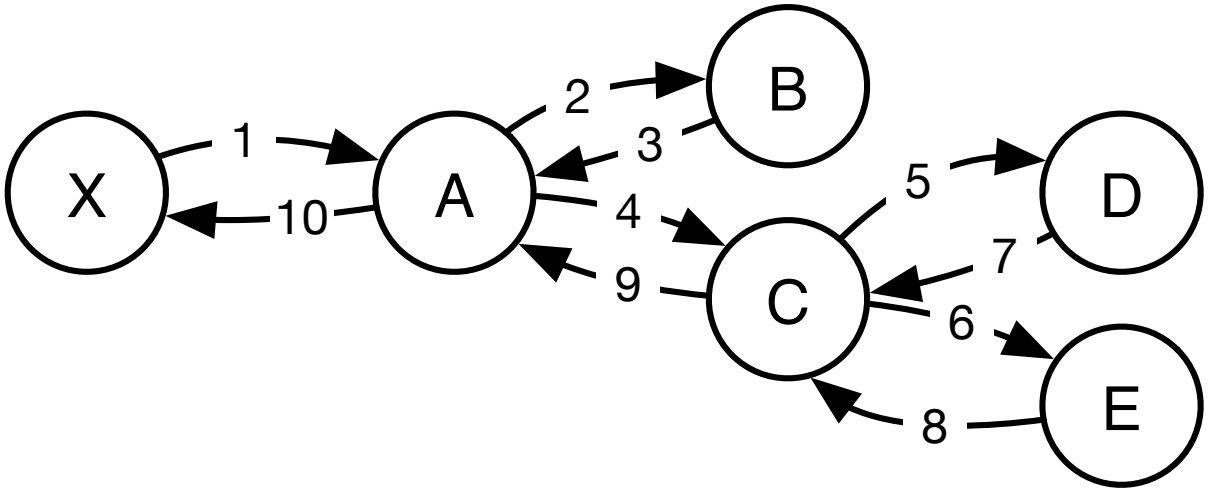

Tracing protokolliert Pfad einer Berechnung
Monitor
Inject Annotated Request
N4
(N4, N5, N6, N2, N1)
(N4)
N1
N5
Ergebnis beinhaltet gesamte Kausalkette der Berechnung
(N4, N5, N6, N2)
N3
N7
(N4, N5)
N2
(N4, N5, N6)
N6

# 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 Ouptuts erlaubt kausale Ordnung
- Timestamps erlauben Reproduktion zeitlicher Abläufe

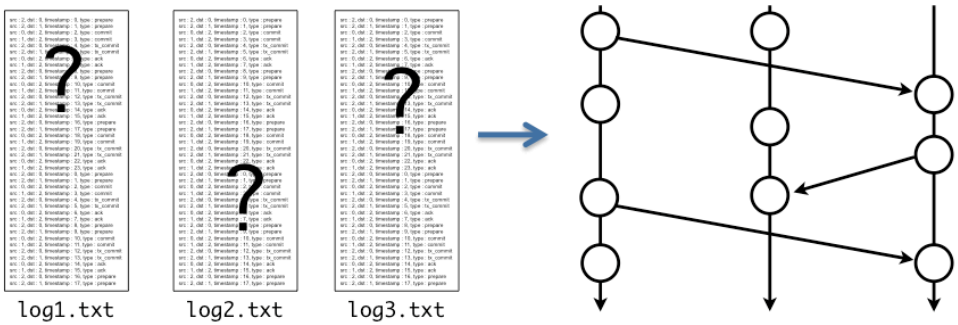
➢ Aussagen über Abhängigkeiten und Berechnungsdauer

# 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 *) [10]
  - **Visualisierung** der Nachrichtenflüsse (z.B. ShiViz)
- **Whitebox-Ansätze** (Quellcode-Ebene):
  - Erfordert streng **strukturiertes Log-Format**
  - Visualisierung aufgezeichneter **Kontroll- und Nachrichtenflüsse** (z.B. Causeway)

* Weitere Informationen in Wei Xu et al.

# 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

Grafik übernommen aus https://bitbucket.org/bestchai/shiviz/wiki/Home

# ShiViz GUI

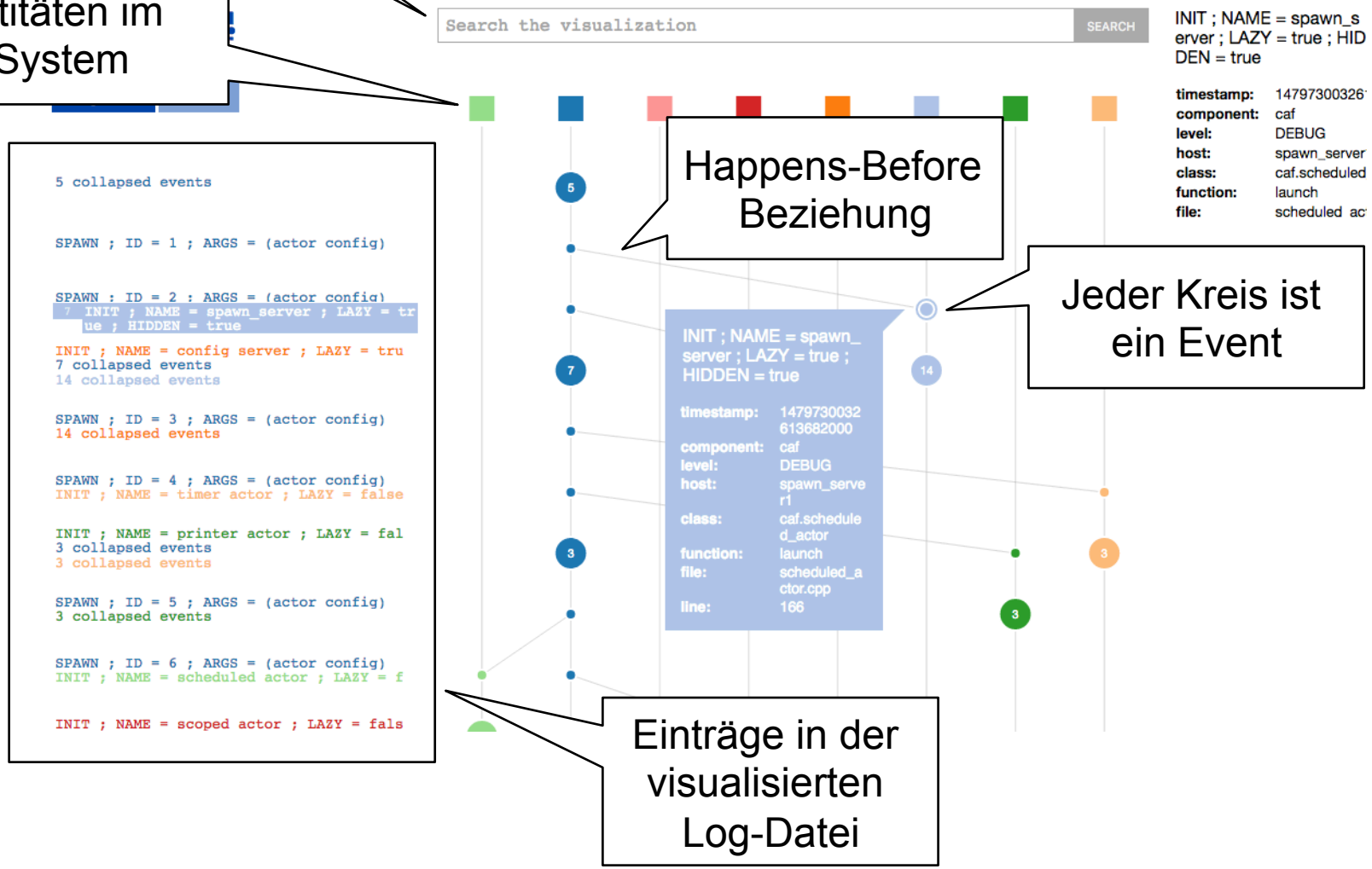
Volltext- und Struktursuche

Entitäten im System

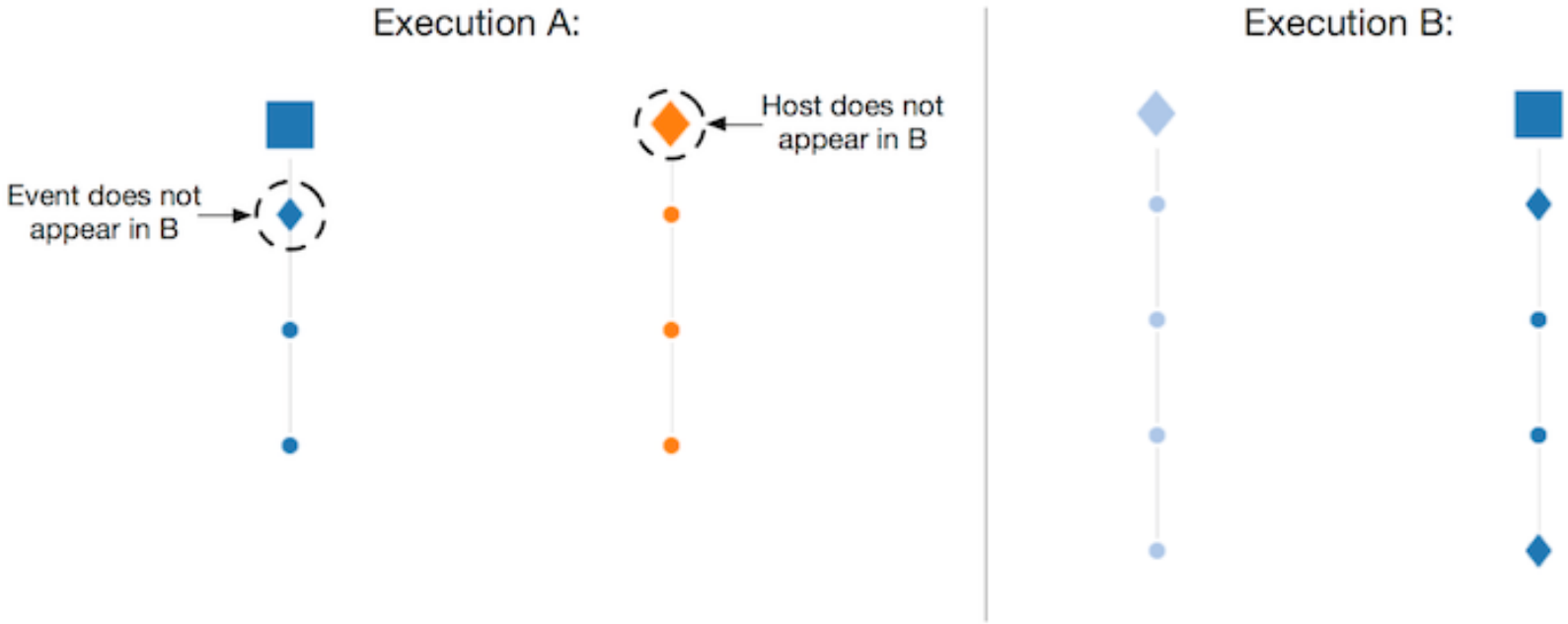
Happens-Before Beziehung

Jeder Kreis ist ein Event

Einträge in der visualisierten Log-Datei



# ShiViz Visual Diff



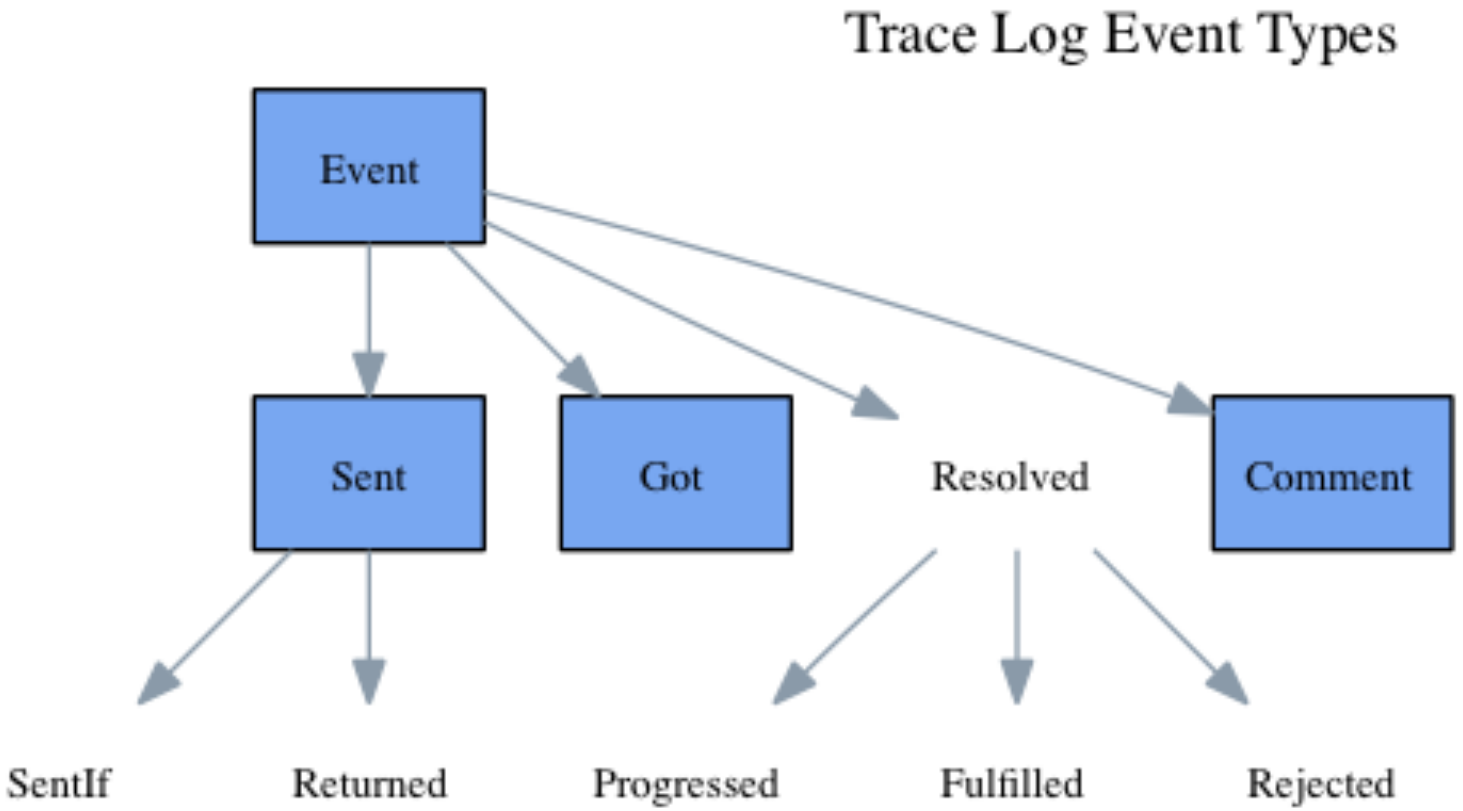

- Per-Event vergleich zweier Programmläufe
- Events/Hosts die nur in einem der verglichenen Läufe vorkommen sind symbolisiert mit ◆
- Erlaubt schnelles auffinden divergierender Abläufe

Grafik übernommen aus http://bestchai.bitbucket.org/shiviz/

# 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

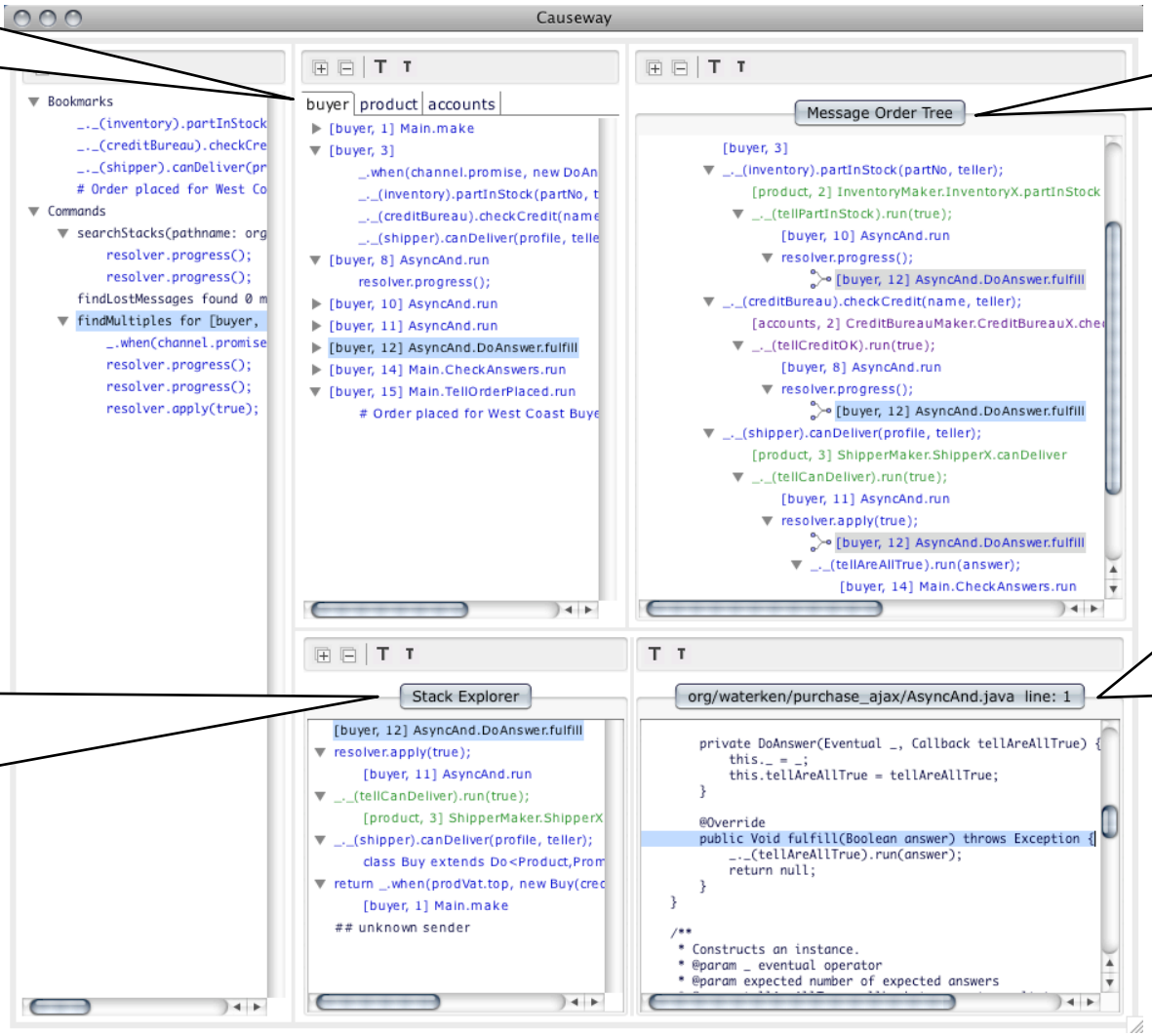



Grafik übernommen aus http://wiki.erights.org/wiki/Causeway_Platform_Developer

# 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

Grafik übernommen aus http://wiki.erights.org/wiki/Causeway

# 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 Gültigkeitsräume verarbeiteter Daten
- **Kontinuierliche Überprüfung** währen der Laufzeit
  - Vor und nach dem Verarbeiten von Daten
  - Bei Zustandsübergängen eines Teilsystems
- **Fehlerbehandlung** bei Verletzung deklarierter Invarianten
  - Fallback: „Selbstheilung“ durch festgelegte Übergänge in sichere Zustände
  - Debugging: Anhalten der Software zur Inspektion oder Abbruch mit aufgezeichneter Fehlerursache

# $D^3S$

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

* Weitere Informationen in Xuezheng Li et al.

# D$^3$S 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)
```

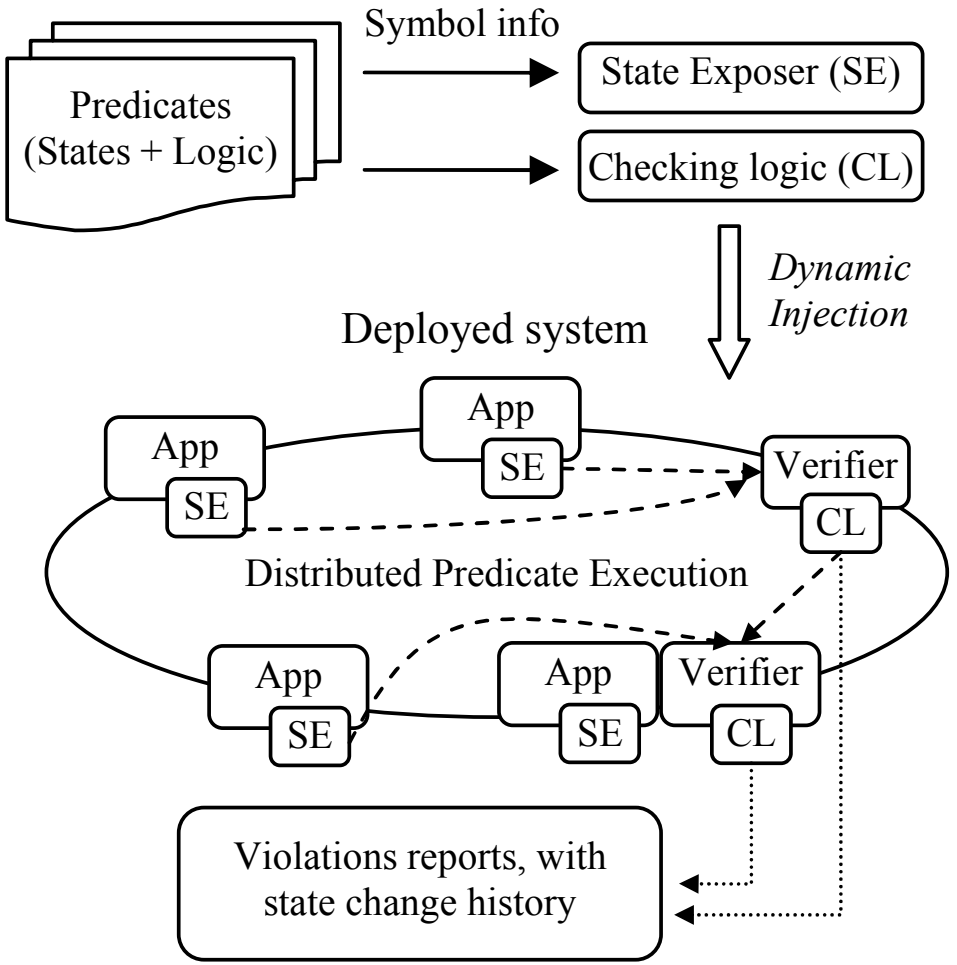
Part 1: define the dataflow and types of states, and how states are retrieved

```
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

Beispiel übernommen aus https://courses.engr.illinois.edu/cs525/sp2010/D3S.pptx

# D³S Architektur




Grafik übernommen aus Xuezheng Li et al.

# Validierung

- **Ziel**: „**fehlerfreie**“ (gemäß Spezifikation) **Software**
- **Vollständige Überprüfung** während der Entwicklungszeit
- **Mathematische Modellierung** aller spezifizierten Systemeigenschaften (Funktionalität, Invarianten, etc.)
- **Formale Spezifikationssprachen** auf Grundlage diskreter Mathematik, Mengentheorie und Prädikatenlogik
- Durch vergleichsweise hohen initialen Aufwand i.d.R. bei **unternehmenskritische Kernkomponenten** angewendet (z.B. Amazon Web Services *: S3, DynamoDB, EBS)

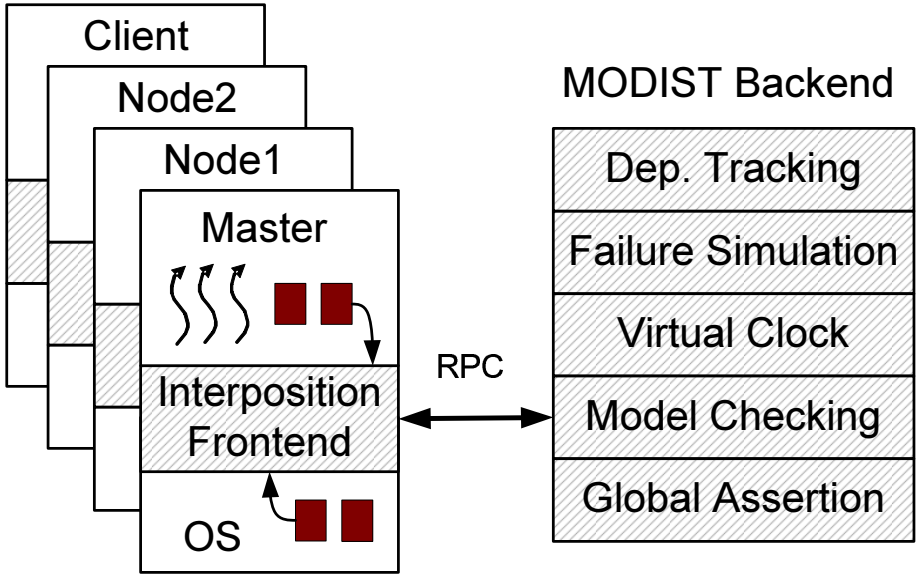
* Weitere Informationen in Chris Newcombe et al.

# 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 Kommunikationskanälen
  - Ausführungen „symbolisch“ anhand des Modells
- **Explicit-State**:
  - Kontrolliertes Ausführen des tatsächlichen Programms
  - I.d.R. nur bis zu fest definierter Ausführungstiefe

# MoDist

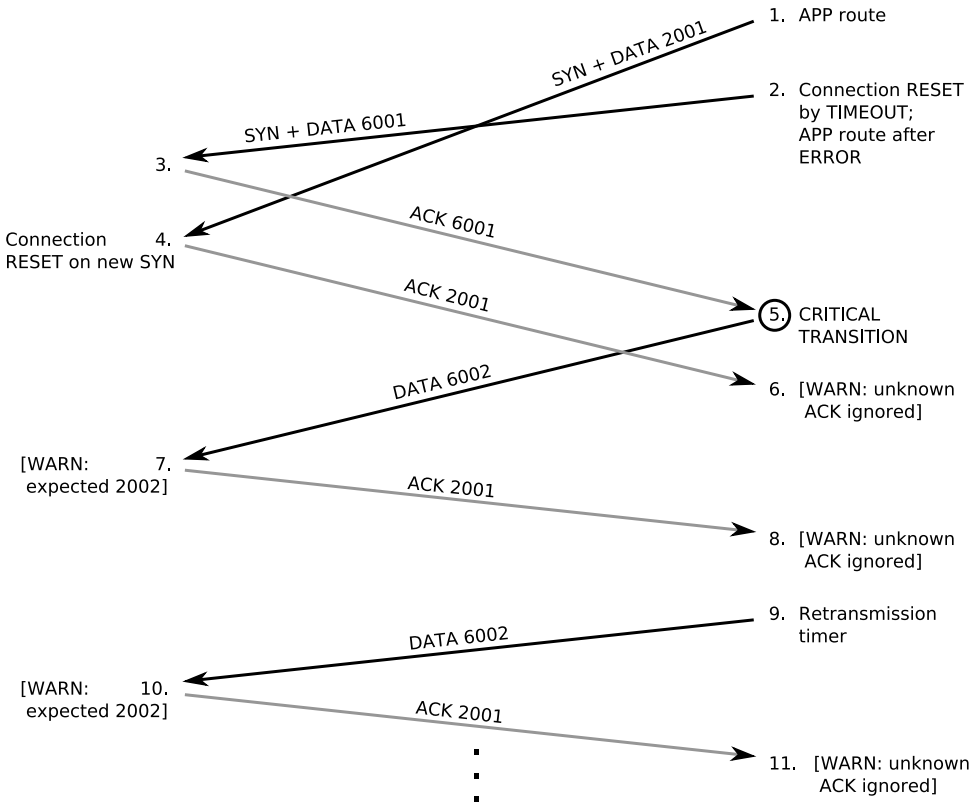
- Blackbox Ansatz
- Transparentes Model Checking von unveränderten Systemen
- Ausführ-Engine zwischen Betriebssystem und Anwendung
- Simulationsumgebung zur deterministischen Ausführung verteilter Anwendungen mit virtueller Uhr




Grafik übernommen aus Junfeng Yang et al.

# 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




Grafik übernommen aus Charles Killian et al.

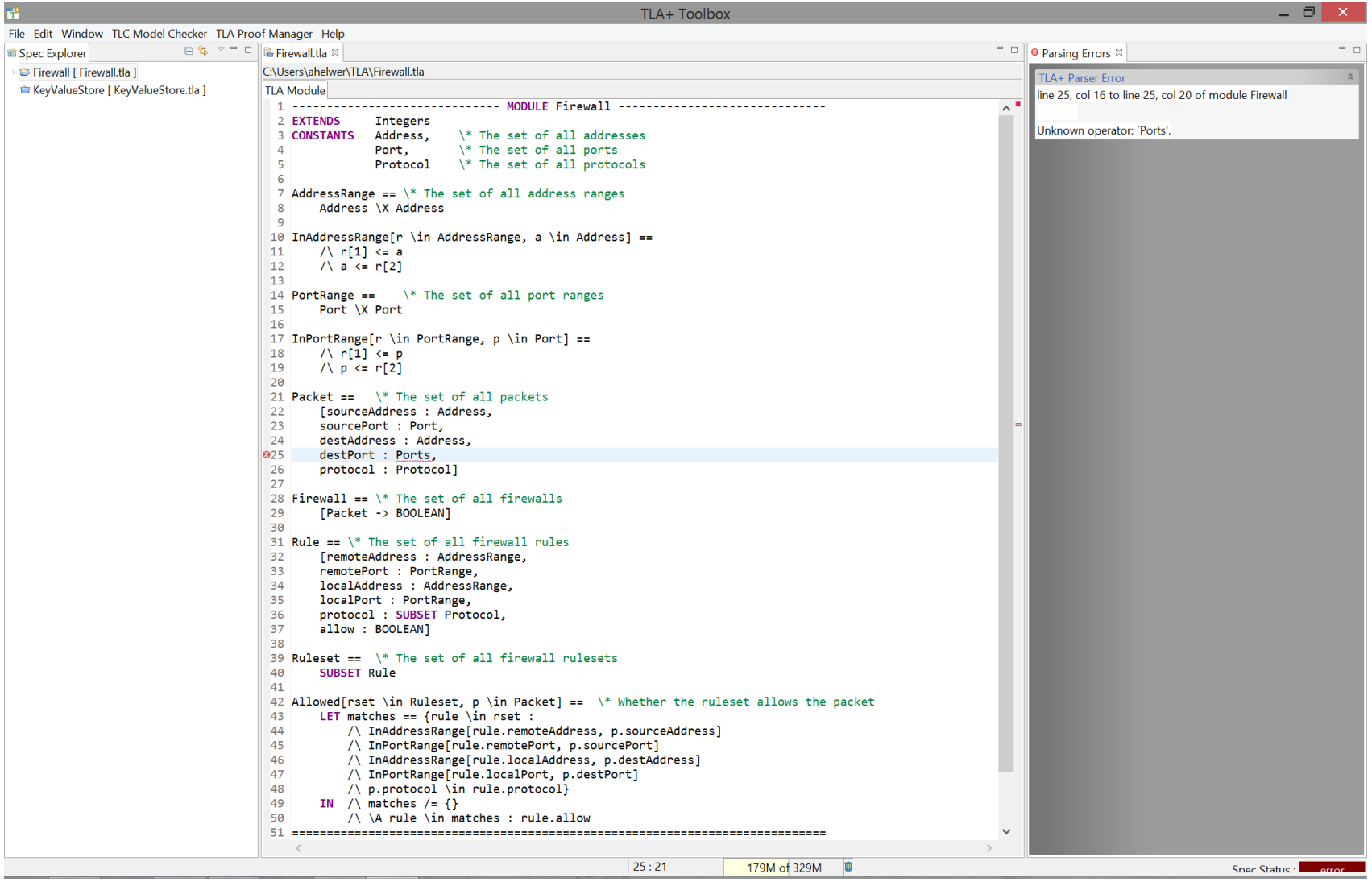
# Theorem Proving

- **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




Grafik übernommen aus https://en.wikipedia.org/wiki/TLA%2B

# 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

# Praktische Herangehensweise

- **Weiterentwicklung** bestehender Software:
  - Langsame Migration hin zu nachrichtenbasierter Programmierung und Middleware (MW)
  - Kapseln bestehender Komponenten
    - Identifikation unabhängiger Programmteile
    - Isolation durch nachrichtenbasierte Fassaden
  - Erweiterung vorhandener Tests
    - Anbindung an Netzwerk-Simulationsmodus der MW
    - Testen der gekapselten Komponenten und deren Zusammenspiel

# Werkzeugeinsatz

- **Visualisieren** von verteilten Systemen erlaubt schnelleres Verständnis von komplexen Zusammenhängen
  - Auf **bestehende Software** anwendbar
  - Evtl. **Anpassung des Log-Formates** (z.B. bei 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
  - Durch **Sampling** auch **als Administrationswerkzeug** interessant

# Werkzeugeinsatz (2)

- 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 $D^3S$
  - Besonders von Interesse bei Software die vom Entwicklerteam auch betrieben wird (z.B. **Microservices**)

# 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.