Verteilte Systeme

Verteiltes Debugging

Fehlerkategorien und –arten bei der verteilten Ausführung

Warum verteilte Ausführung?

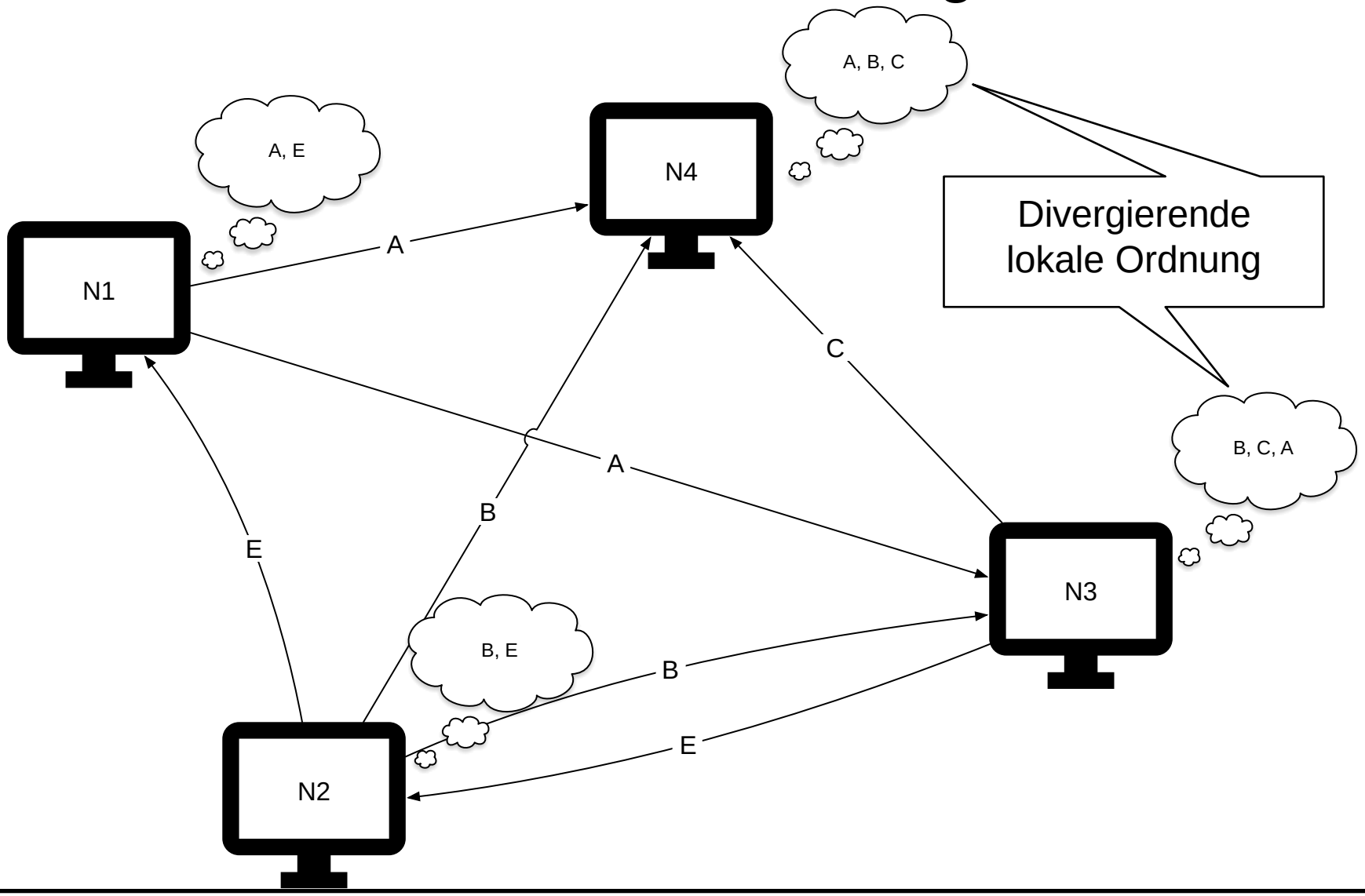
- **Skalierbarkeit:**
  - **Hohe Anfrage** an einen Dienst **übersteigt die Ressourcen** einer einzelnen Maschine
  - **Speicherbedarf** vieler Anwendungen (siehe Google, Facebook, etc.) übersteigt das Maximum einzelner Server
  - **Software muss** mit der Anzahl an Benutzern „**mitwachsen**“ können, um marktgerecht zu sein

- **Robustheit:**
  - Der **Ausfall** einzelner Komponenten muss **kompensierbar** werden
  - Infrastruktur-Software muss **hochverfügbar** sein **durch Redundanzen und Fehlertoleranz**

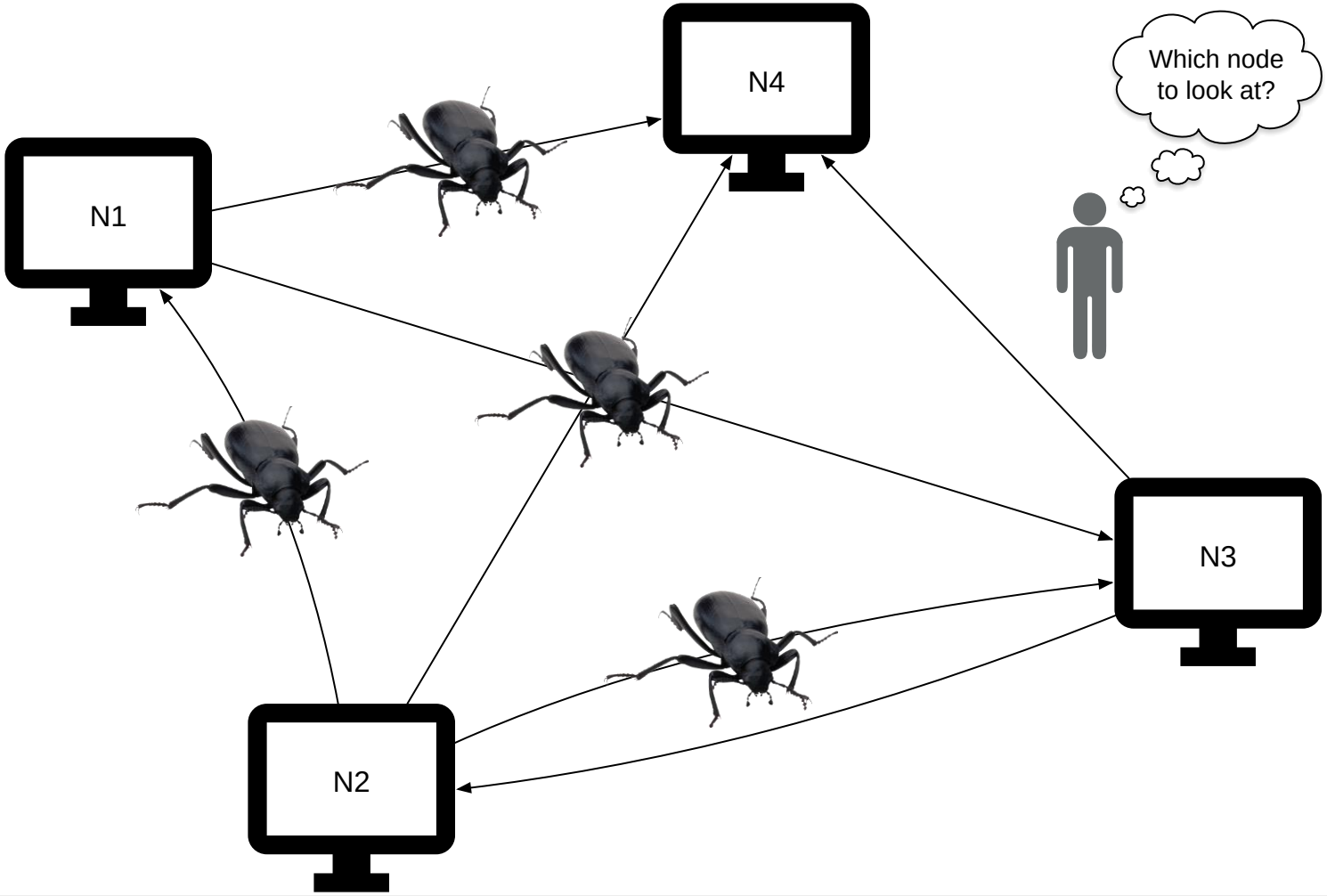
Eigenschaften verteilter Ausführung

- **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
- **Eng verzahnte Logik** für Funktion und Fehlerbehandlung
  - 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

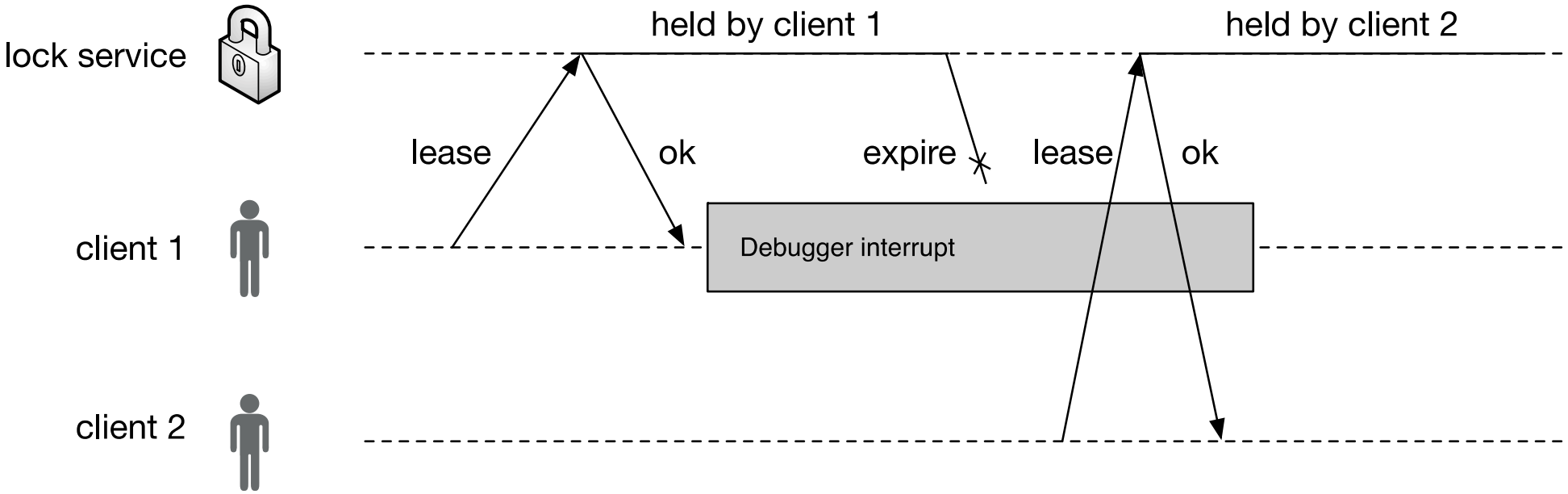
- **Programmierwerkzeuge** haben großen Einfluss
  - Komplexität der Fehlerbehandlung kritisch
  - Ungünstige Abstraktionen verwischen Fehlerursachen
- **OO: viele Abhängigkeiten** und Abstraktionsschichten
  - Proxy-Objekte und RMI „verbergen“ Verteilung
  - Interagierende Objekten oft nicht für Verteilung 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



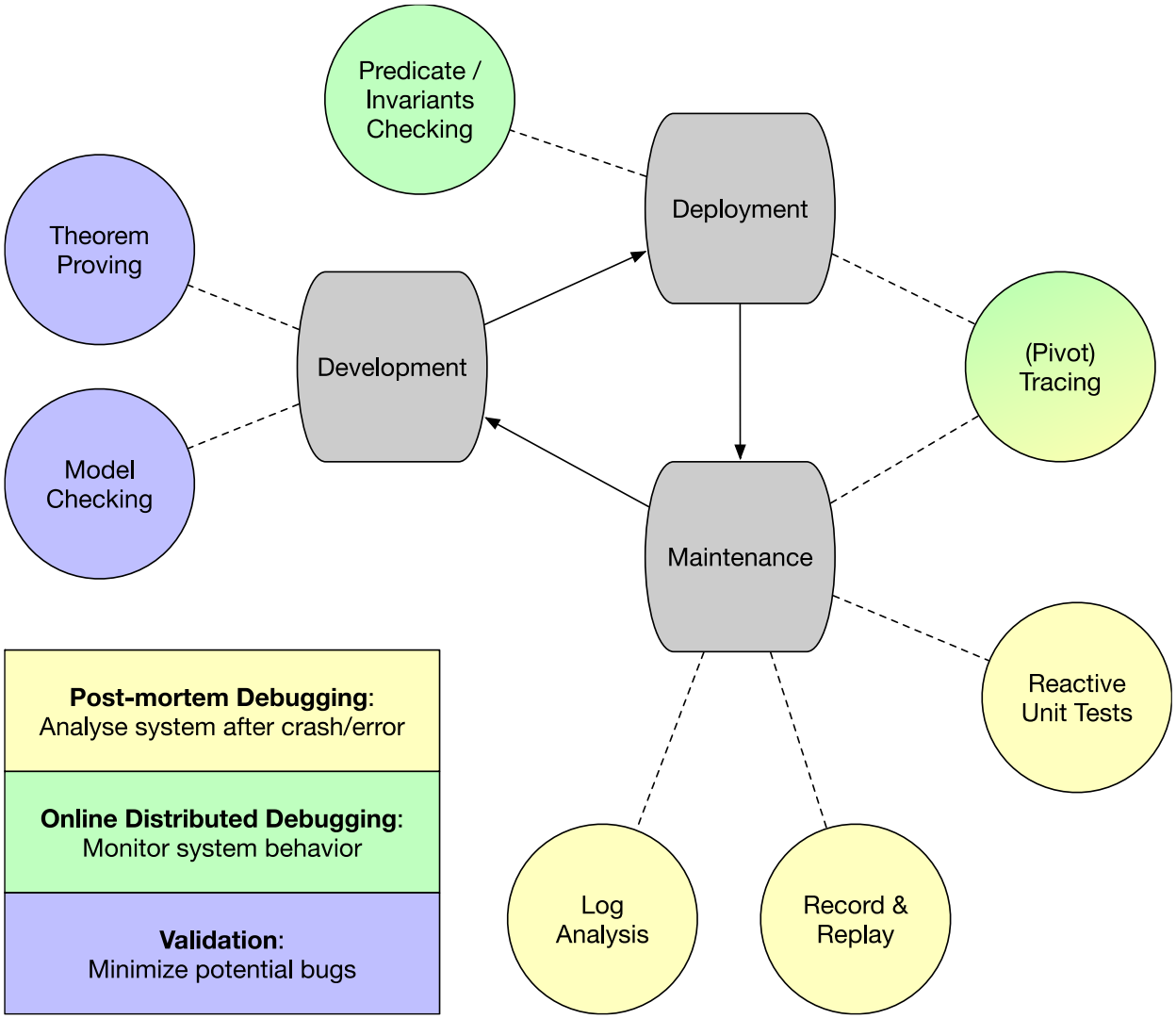
- Lock Service unabhängig von Clients
- Debuggen von Client 1 während des Lease schwierig:
  - Ausbremsen von Client 1 führt zu Lock-Verfall
  - Andere Clients nicht kontrolliert durch Debugger

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**: Auswertung 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

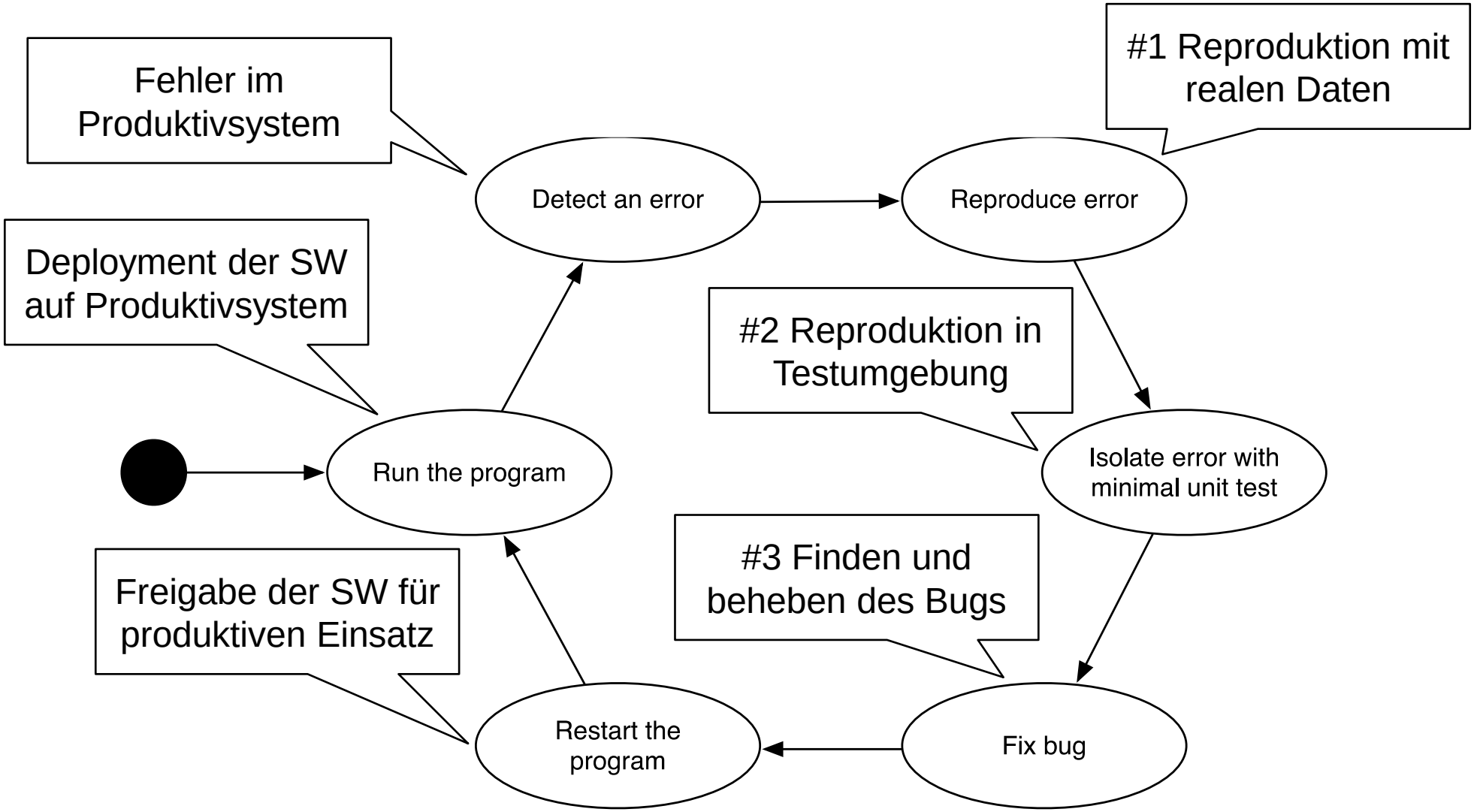
Post-mortem Debugging

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**

* Weitere Informationen in Ding Yuan et al.

Typischer Kreislauf reaktiver Tests



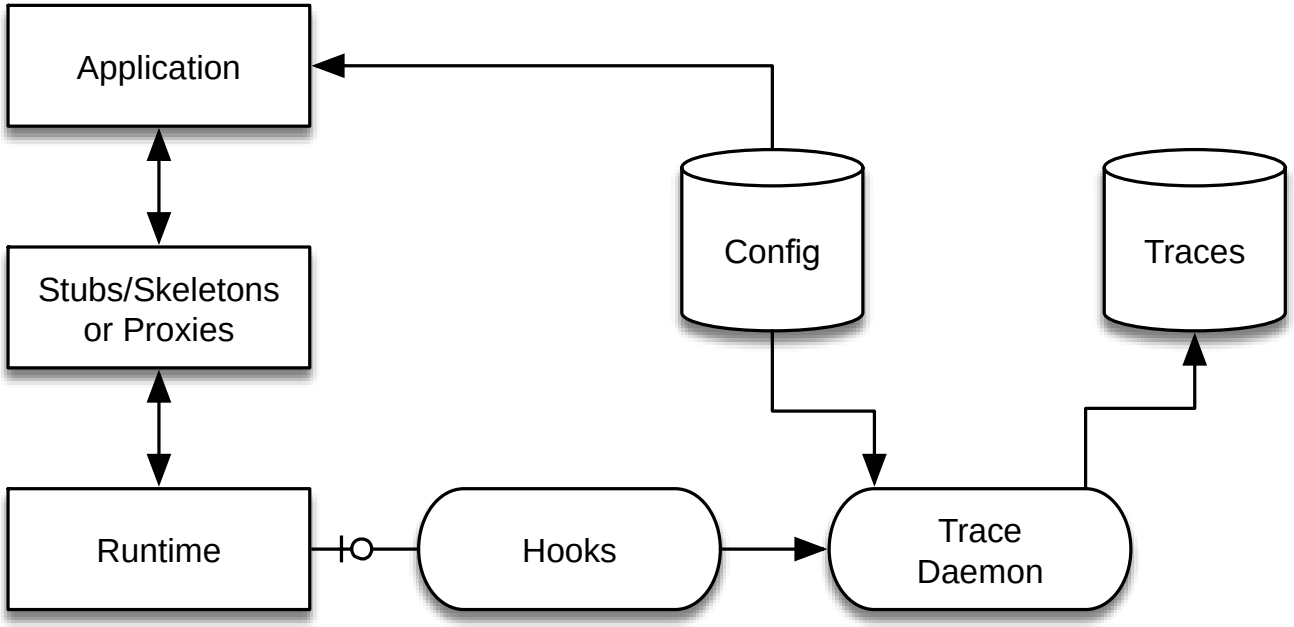
Record and Replay

- 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 *

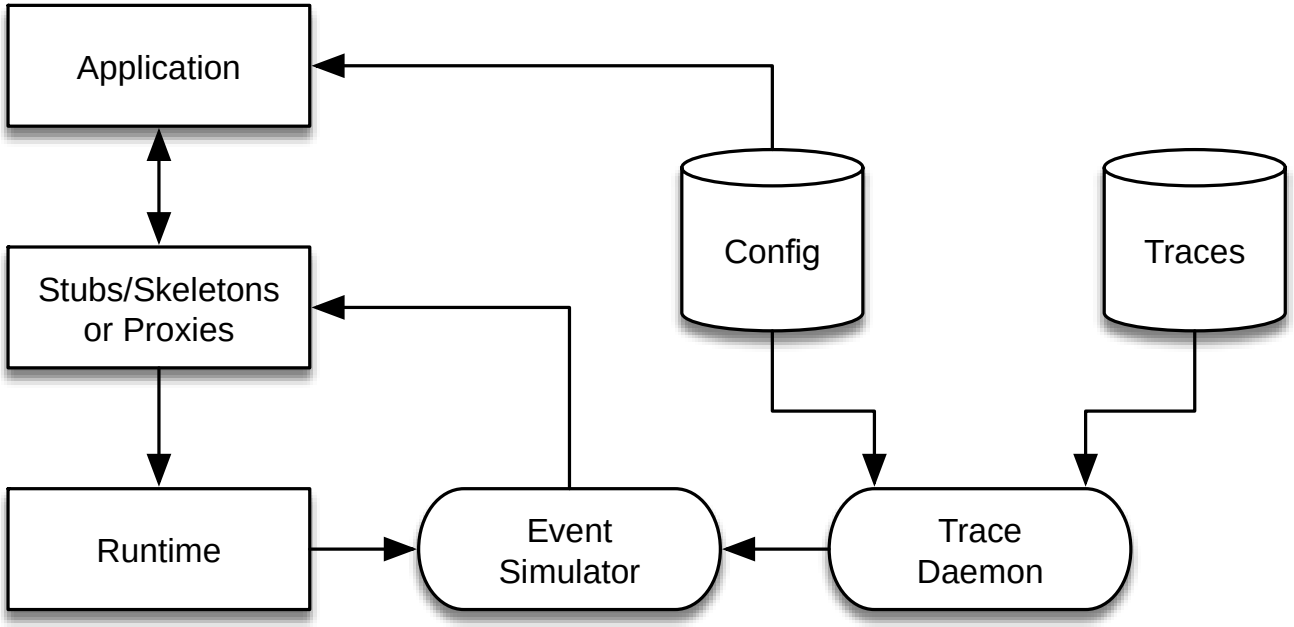
* Weitere Informationen in Dennis Geels et al.

Record Phase



- Anwendung kommuniziert über Proxy-Objekte zu Remotes
- Aufzeichnen aller Netzwerk- und I/O-Events über Hooks
- Speichern des Ablaufs 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)

Mozilla rr *

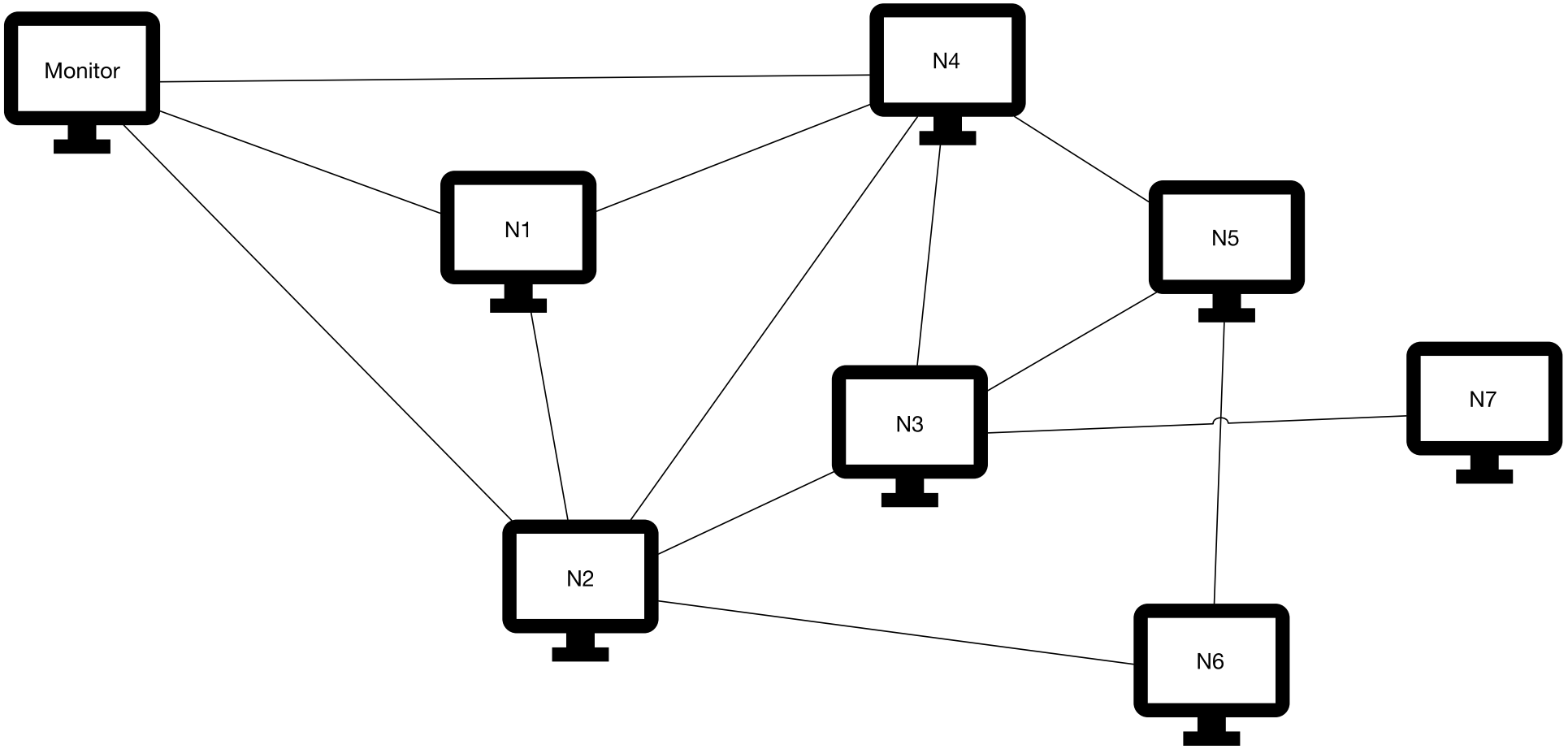
- GDB-Replacement/Erweiterung
- Erlaubt Analyse nichtdeterministischer Programmfehler
- Record-Phase: `rr record <command>`
  - Führt Programm in simulierter Single-Core Umgebung aus
  - Optional: Scheduler Optionen & Driver
    - Größere Abdeckung des Zustandsraums
    - Aggressive Context-Wechsel lösen *Races* häufiger aus
- Replay-Phase: `rr replay`
  - Startet letzte Aufzeichnung in GDB
  - Vollständig deterministischer Programmablauf:
    - Befehlsfolge und Ergebnisse *aller* Syscalls fixiert
    - Speicheradressen und Registerinhalte stets gleich

* Siehe https://rr-project.org, Quellcode online verfügbar unter https://github.com/mozilla/rr

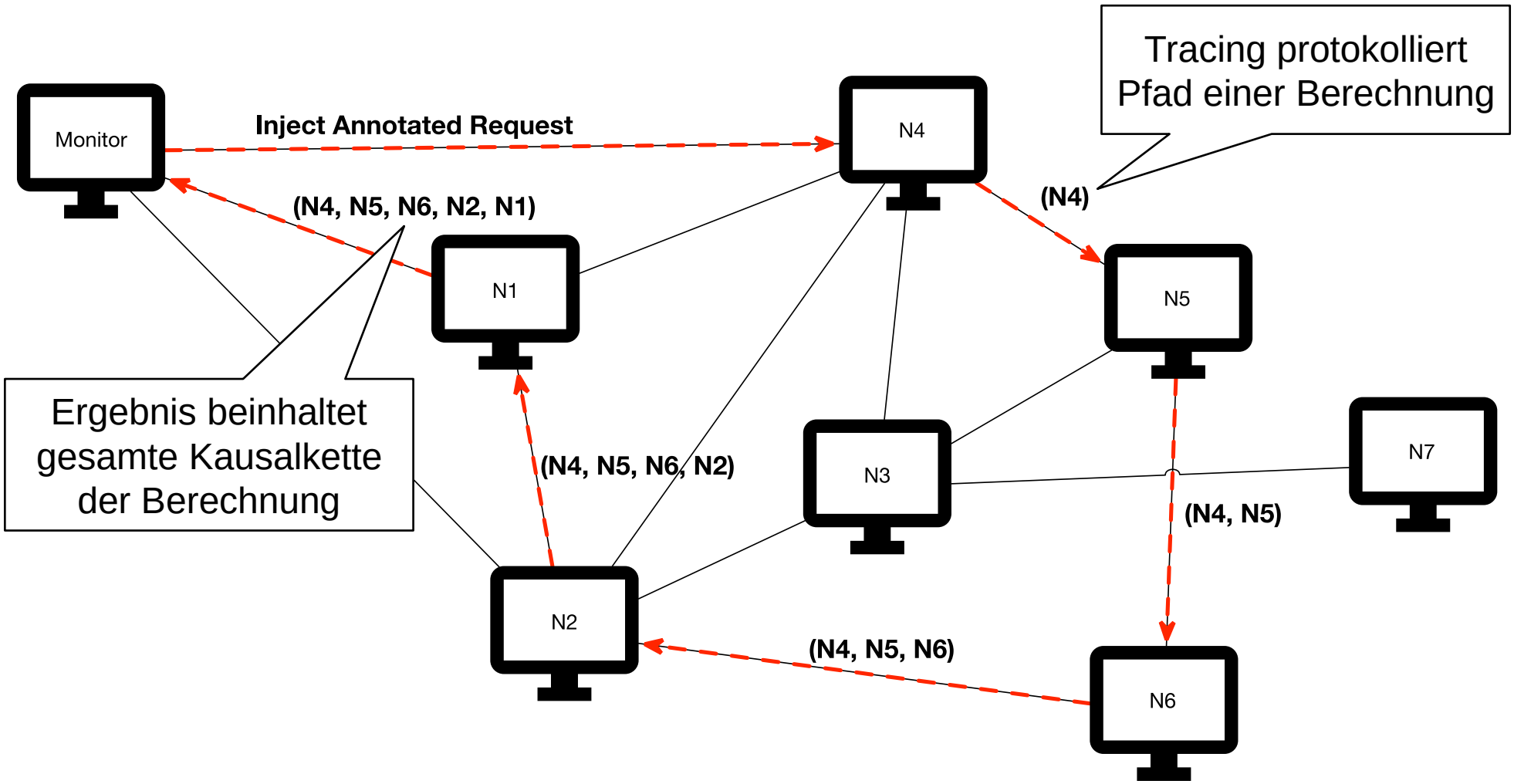
Tracing

- **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

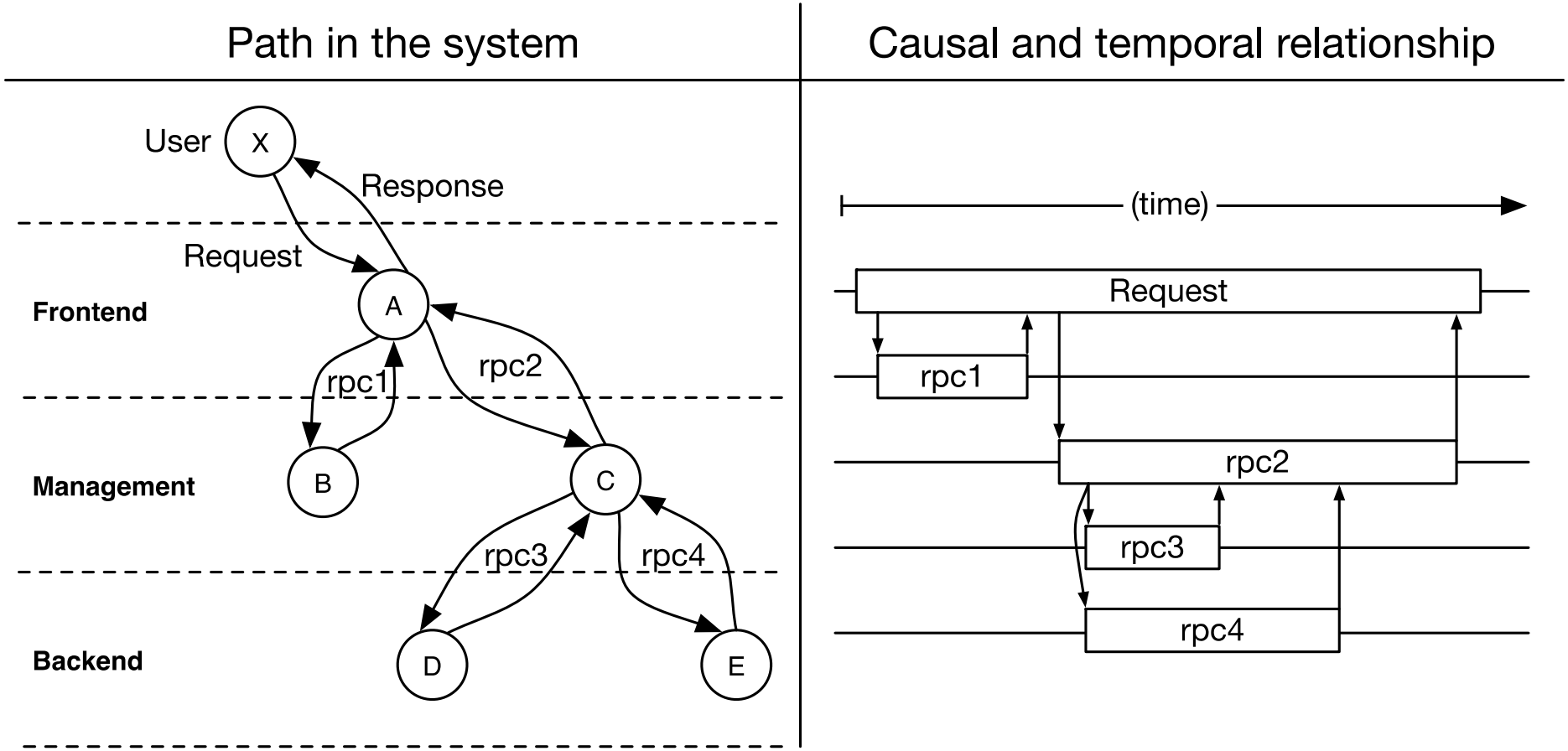
Tracing



Tracing

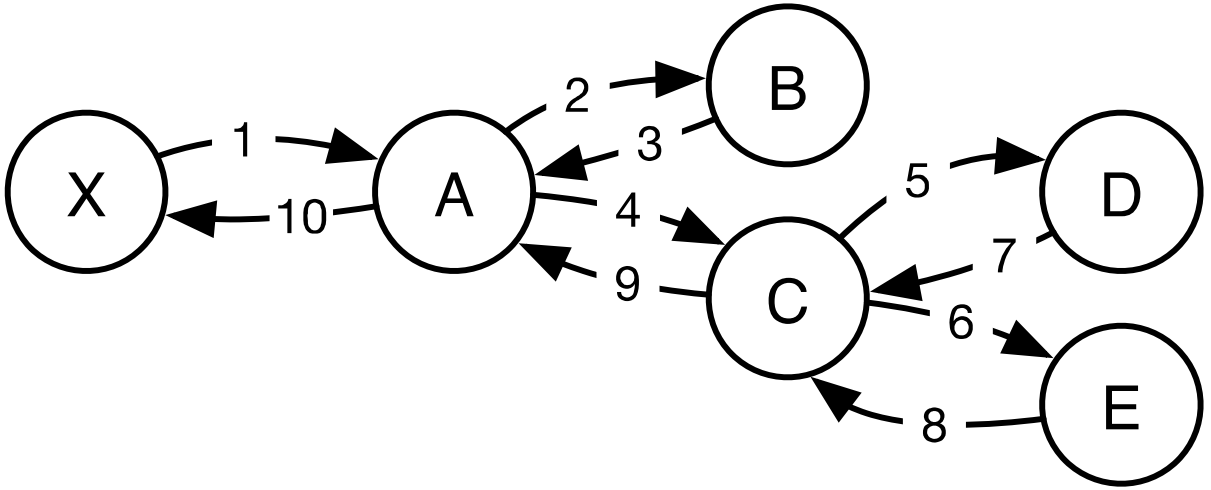

Tracing protokolliert Pfad einer Berechnung
Monitor
Inject Annotated Request
(N4, N5, N6, N2, N1)
N4
(N4)
N1
N5
Ergebnis beinhaltet gesamte Kausalkette der Berechnung
(N4, N5, N6, N2)
N3
N7
(N4, N5)
N2
(N4, N5, N6)
N6

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 Outputs erlaubt kausale Ordnung
- Timestamps erlauben Reproduktion zeitlicher Abläufe

- Aussagen über Abhängigkeiten und Berechnungsdauer

OpenTelemetry *

- Standardisierte API für herstellerneutrale Instrumentierung
- Verfügbar für Python, Java, Go, C++, ...
- Ein Trace ist ein DAG aus *Spans* die über eine *Context* eine Beziehung aufhaben
  - Spans haben Metadatan: *parent Span, Attribute, etc.*
  - Ein *Context* kann zwischen Services propagiert werden
- Ein *Tracer* erzeugt *Spans* und vergibt Namen
  - Namen identifizieren Operationen, z.B. `get_user`
  - Bei RPC: Name entspricht i.d.R. dem Methodenname
- Tracing-Kontext muss in System-Events eingebettet werden
  - Einbettung in HTML via zusätzliche Header-Informationen
  - Dafür gibt es W3C Trace Context HTTP headers
- Daten können über OTLP (OpenTelemetry Protocol) exportiert werden
  - Verarbeitung z.B. mit Jaeger (https://www.jaegertracing.io)

* Siehe https://opentelemetry.io

Log-Analyse

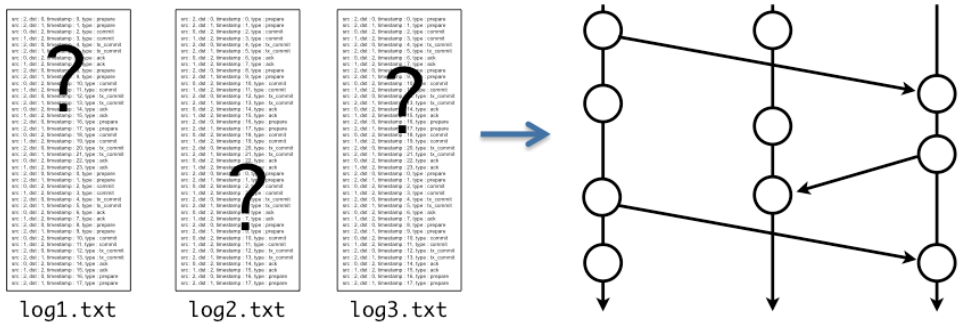
- **Auswerten von Konsolen-, Debug- oder Systemlogs**
  → i.d.R. bei beliebiger Software *ohne Änderung* möglich
  - **Leichtgewichtig**, aber oft zu detailliert ohne Tool-Support

- **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**

* Weitere Informationen in Wei Xu et al.

ShiViz



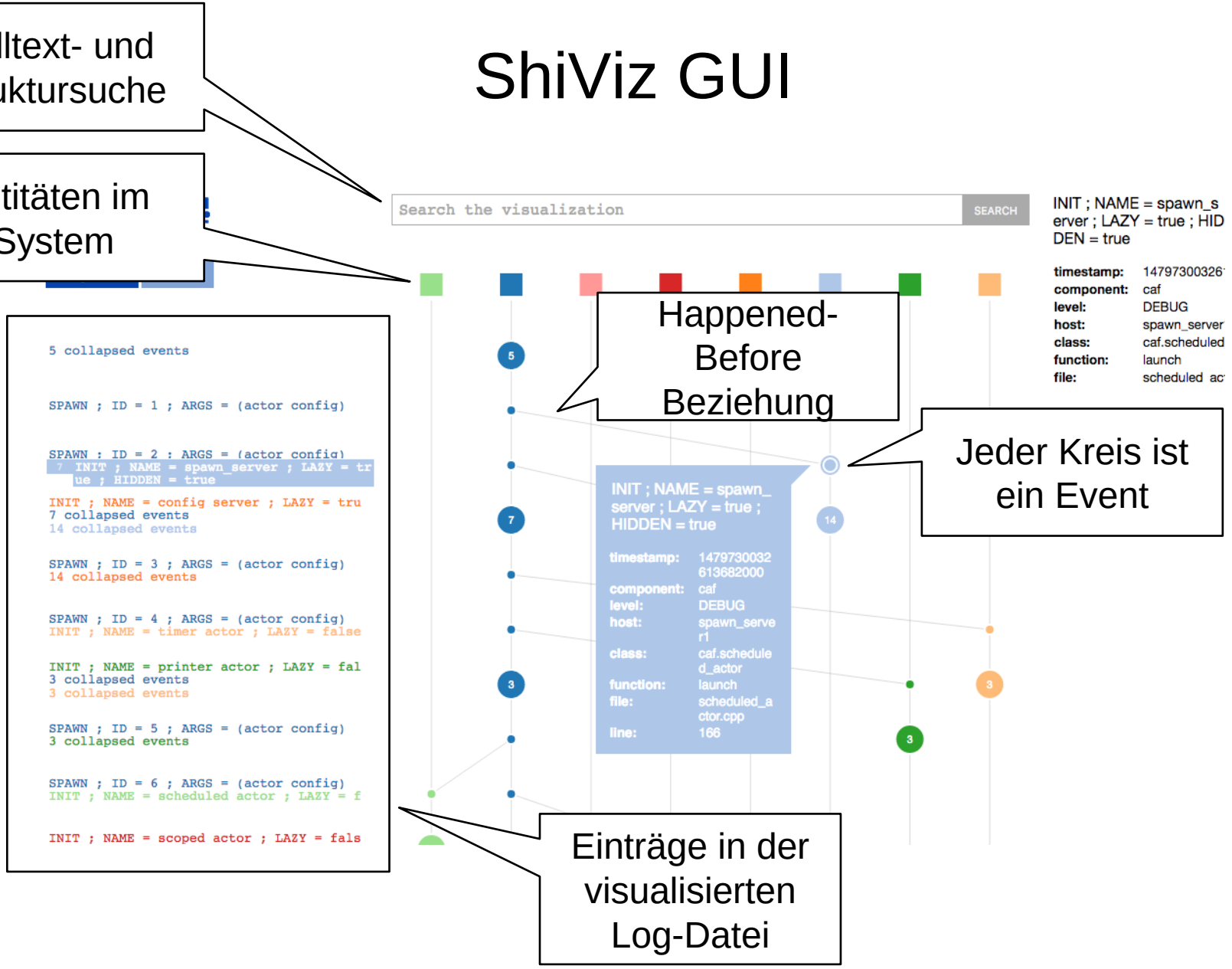
- **Visualisierung** von Log-Dateien **als interaktive Kommunikationsgraphen** mit kausaler Ordnung
- Import **beliebiger Log-Formate** (Regex-basierter Importer)
- Anforderung: JSON-formatierte **Vector-Timestamps**
- **Volltextsuche** über Log-Ereignisse **sowie strukturierte Suche** nach Kommunikationsmustern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

Grafik übernommen aus https://github.com/DistributedClocks/shiviz/wiki

ShiViz GUI

Volltext- und Struktursuche

Entitäten im System

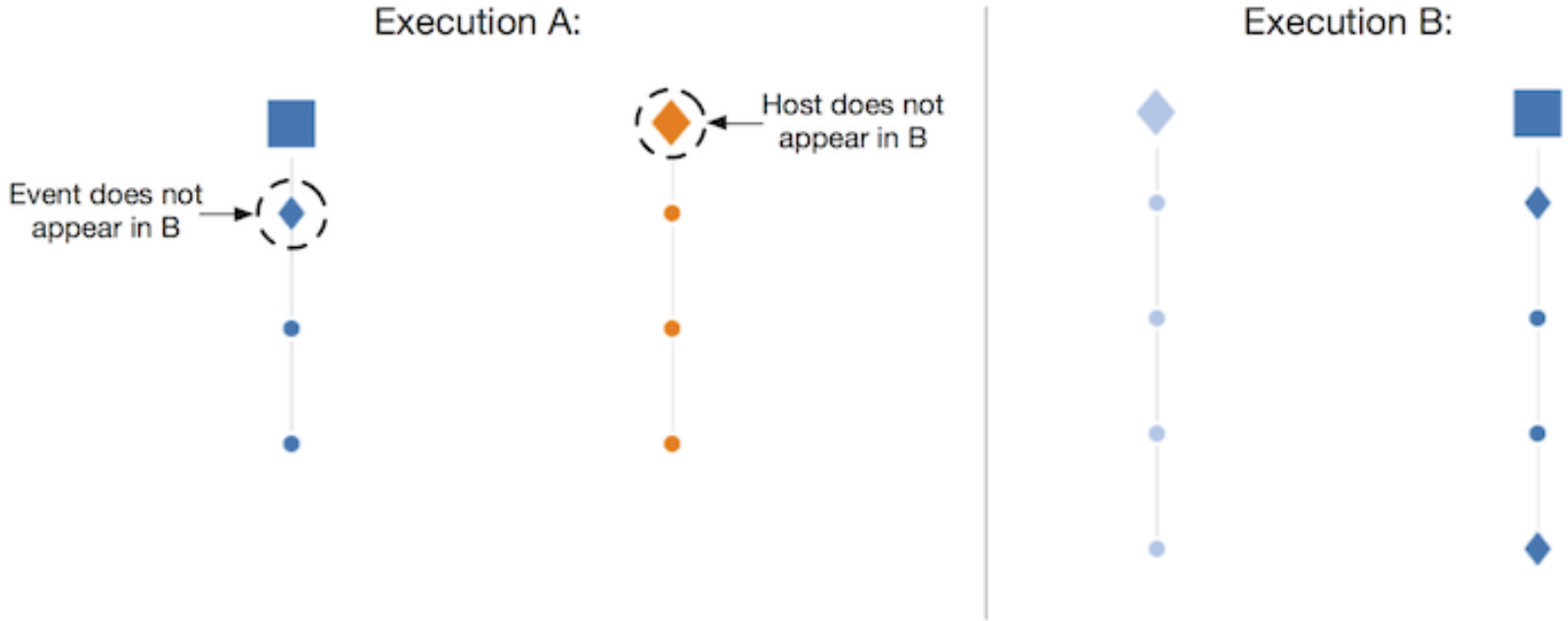


Happened-Before Beziehung

Jeder Kreis ist ein Event

Einträge in der visualisierten Log-Datei

ShiViz Visual Diff



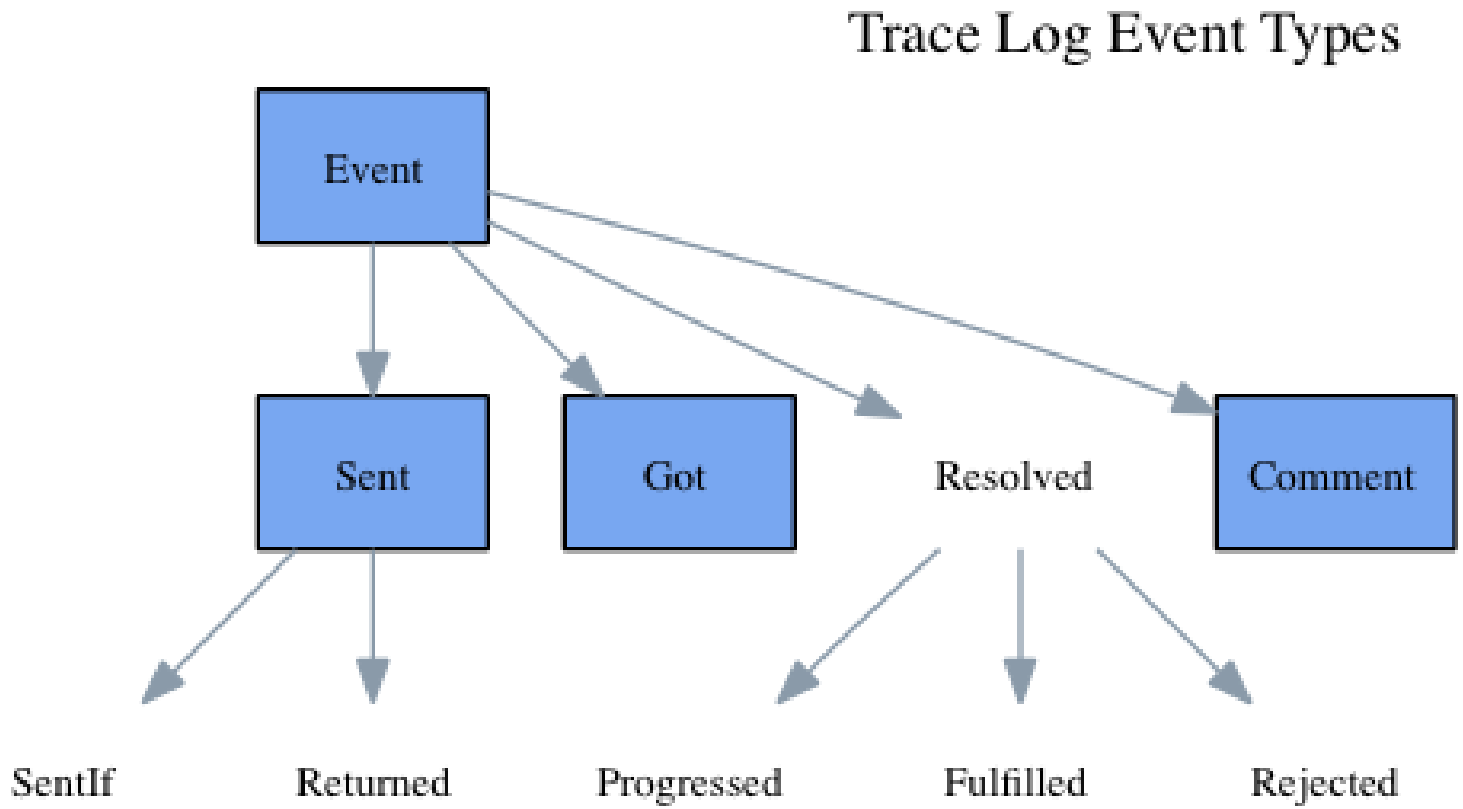
- Per-Event vergleich zweier Programmläufe
- Events/Hosts die nur in einem der verglichenen Läufe vorkommen sind symbolisiert mit ◆
- Erlaubt schnelles auffinden divergierender Abläufe

Grafik übernommen aus https://bestchai.bitbucket.io/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

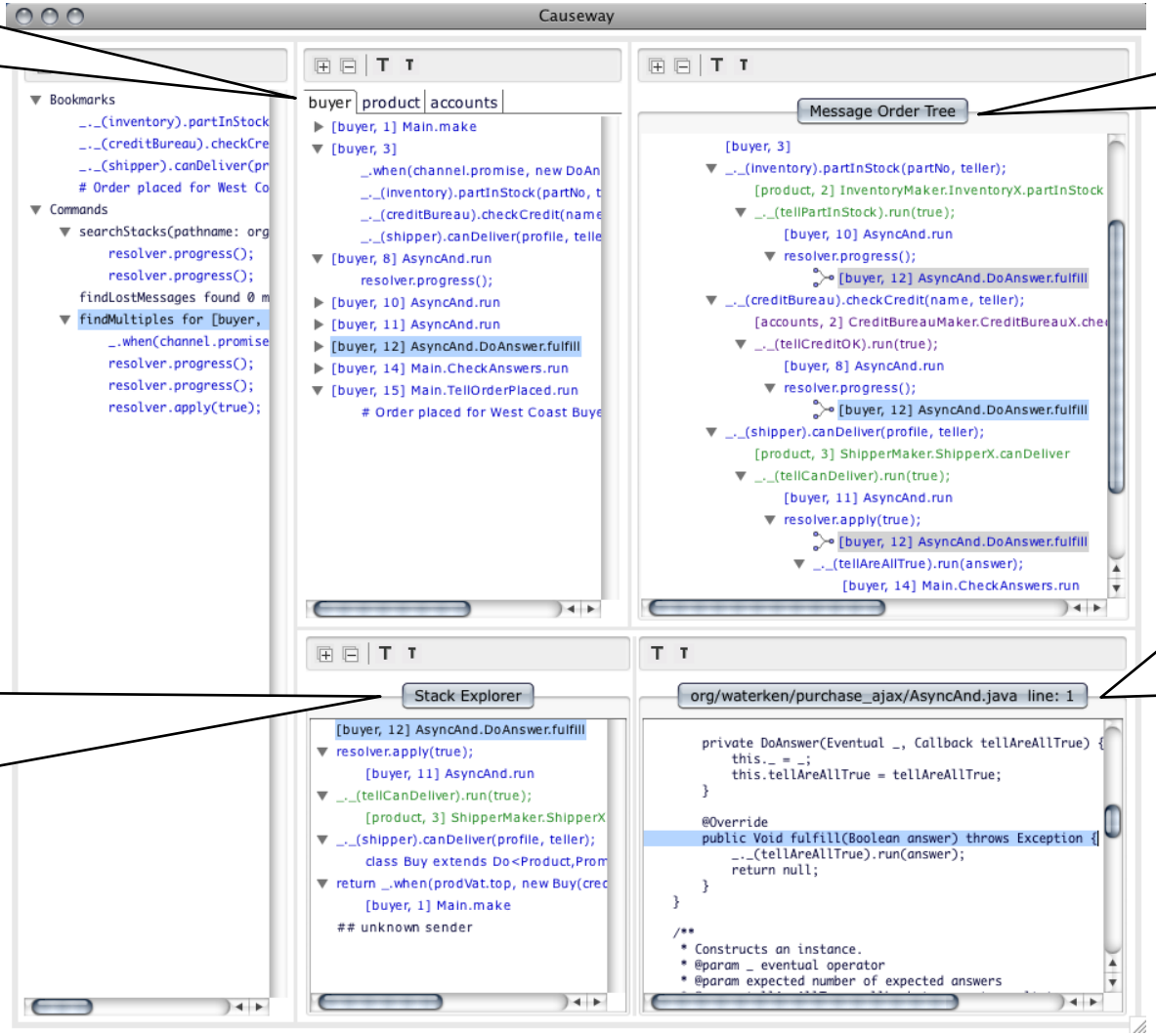


Grafik übernommen aus http://wiki.erights.org/wiki/Causeway_Platform_Developer

Causeway GUI

Entitäten im System

Nachrichten-Flüsse im System

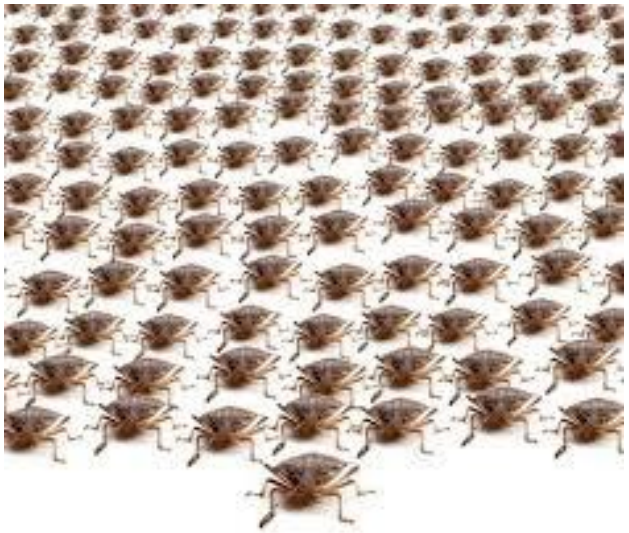


Aktueller Call-Stack zur angezeigten Quellcode-Stelle

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

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

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

* Weitere Informationen in Jonathan Mace et al.

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 deklarierter Invarianten
  - Fallback: „Selbstheilung“ durch festgelegte Übergänge in sichere Zustände
  - Debugging: Anhalten der Software zur Inspektion oder Abbruch mit aufgezeichneter Fehlerursache

D³S: Debugging Deployed Distributed Systems

- **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)

* Weitere Informationen in Xuezheng Li et al.

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)

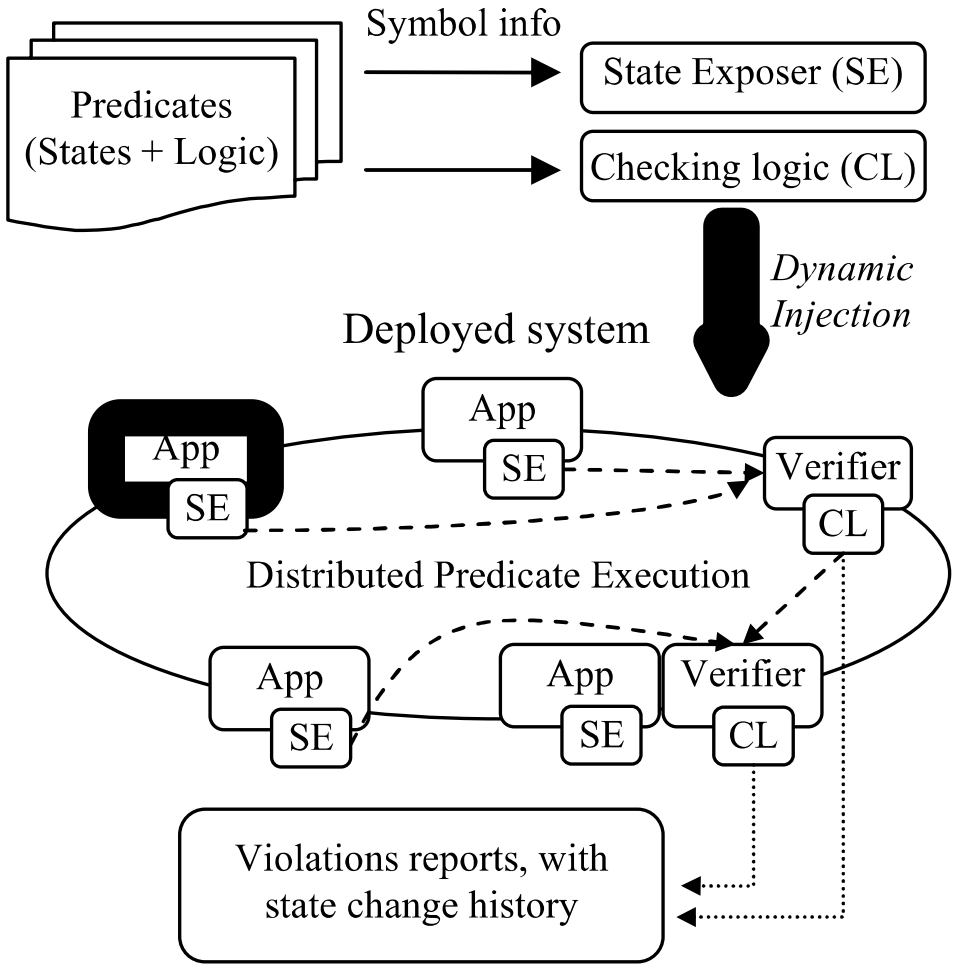
Part 1: define the dataflow and types of states, and how states are retrieved

```
class MyChecker : public 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

Validierung

- **Ziel**: „**fehlerfreie**“ **Software** (gemäß Spezifikation)
- **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 Anfangsaufwand i.d.R. nur bei **unternehmenskritische Kernkomponenten** angewendet (z.B. Amazon Web Services *: S3, DynamoDB, EBS)

* Weitere Informationen in Chris Newcombe et al.

Model Checking

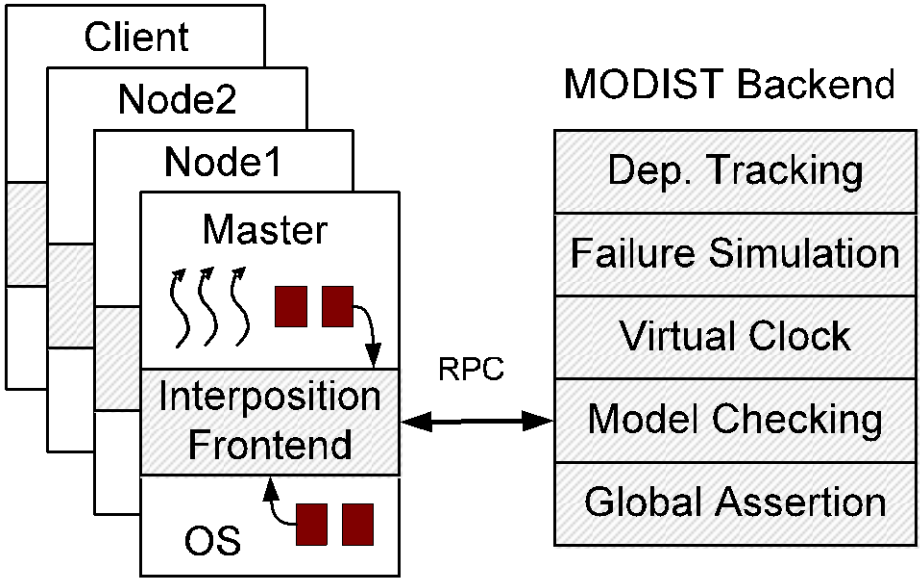
- **Erschöpfendes**, automatisches **Testen** eines Programms
  - **Maschinenlesbare Definition** gültiger Systemzustände, aus denen Testfälle abgeleitet werden
  - **Großer Zustandsraum** limitiert Skalierbarkeit in der Praxis

- **Symbolisch**:
  - **Mathematische Modellierung** des gesamten Systems **als Zustandsautomat** inklusive der Kommunikationskanäle
  - **Symbolische Ausführung** des Modells

- **Explicit-State**:
  - **Kontrolliertes Ausführen** des tatsächlichen Programms
  - I.d.R. nur bis zu **vordefinierter Ausführungstiefe**

MoDist

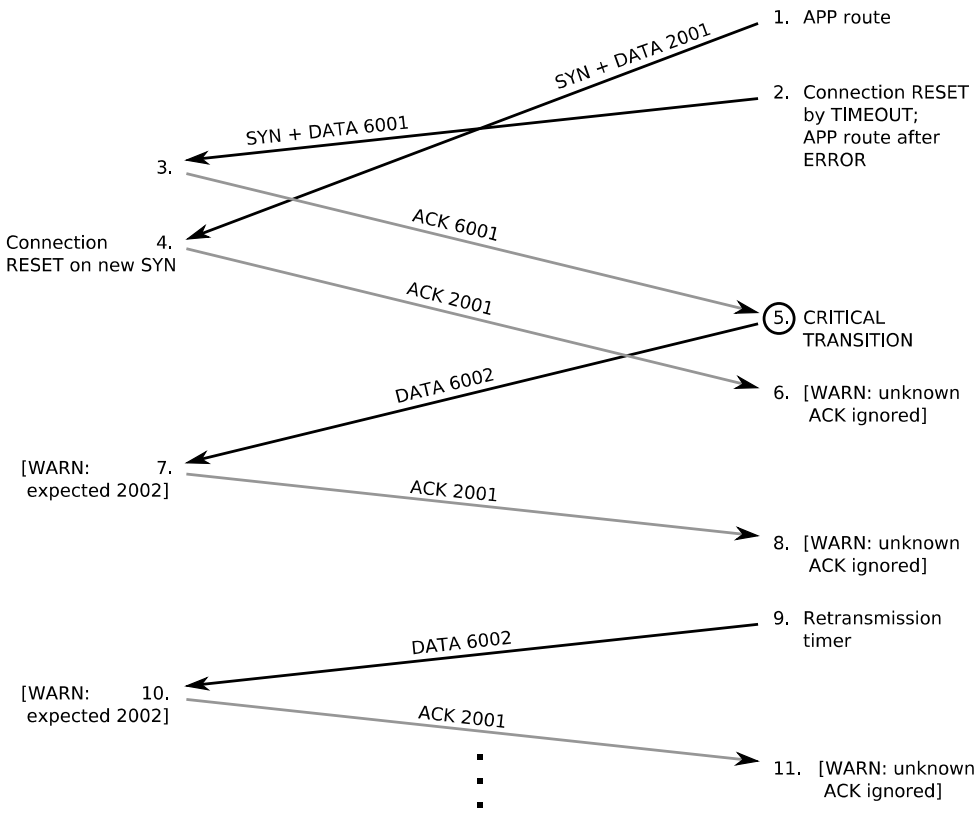
- Analysiert beliebige Anwendungen unverändert als **Blackbox**
- **Ausführ-Engine** zwischen Betriebssystem und Anwendung
- Simulationsumgebung erlaubt **deterministische Ausführung** verteilter Anwendungen mit virtueller Uhr



Grafik übernommen aus Junfeng Yang et al.

MaceMC

- Erfordert Software-Ergänzungen zur **Whitebox**-Analyse
- 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.

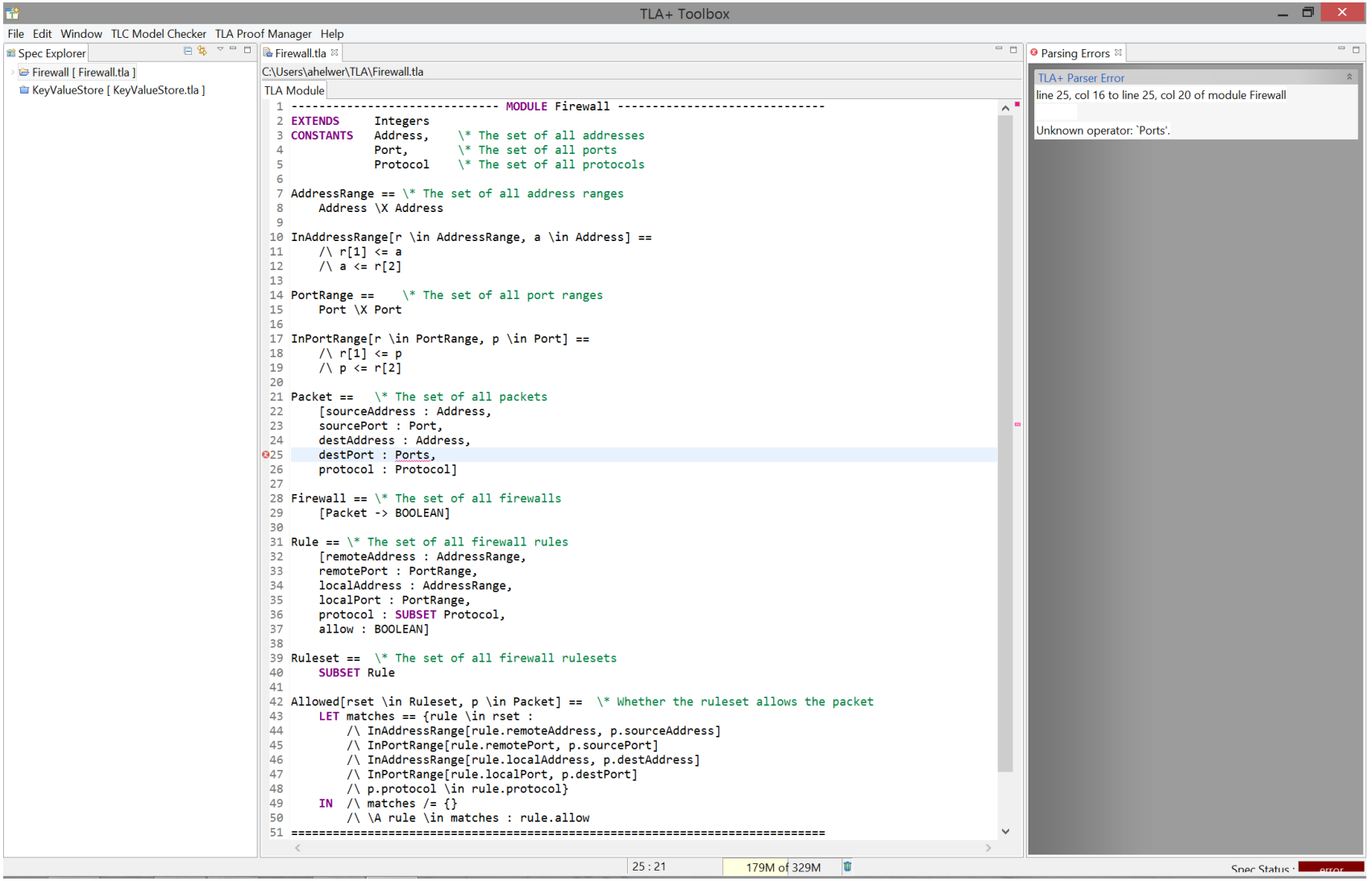
Theorem Proving

- **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 erlauben **automatisiertes Testen** (Model Checking) **entwickelter Implementierungen**

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 (z.B. determistische Test-DSL für CAF)
- **Testgetriebene Entwicklung**
  - Umfassende 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 (z.B. OpenTracing) 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 D^3S
  - 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.