Rapid I/O Toolkit

http://projects.spamt.net/riot

Alexander Bernauer
alex@copton.net

08.12.08

Inhalt

- Motivation
- Architektur
- Beispiel
- I/O
- Features
- Ausblick

Motivation

Problemstellung

- Vorgaben
  - Datenverarbeitung
  - sehr hohe Last
  - Hardware mit parallelen CPU und I/O Resourcen
- Ziel
  - maximaler Durchsatz

Wie?

- effiziente Architektur
  - alle Resourcen ständig (sinnvoll) auslasten
  - wenig Overhead durch Verwaltung
  - wenig Synchronisation
- effiziente Implementierung
  - C++

Hürden

- I/O verursacht blockierende Syscalls
  - Thread wartet
  - CPU läuft leer

Fork Modell

- hoher Verwaltungsaufwand
- Overhead steigt mit der Last
- DoS

Thread-Pools

- zu wenig Threads
  - CPU läuft leer
- zu viele Threads
  - Verwaltungsaufwand beim Umschalten
  - ständig kalte Caches
- dynamische Poolgröße
  - Verwaltungsaufwand

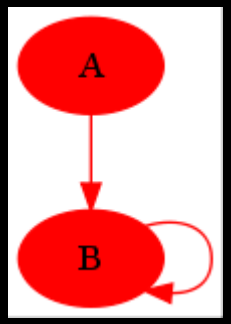
Lösung

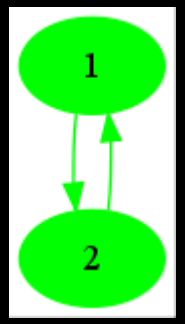
- Ereignisbasierte Architektur
- I/O wird asynchron gemacht

Architektur

Paradigma

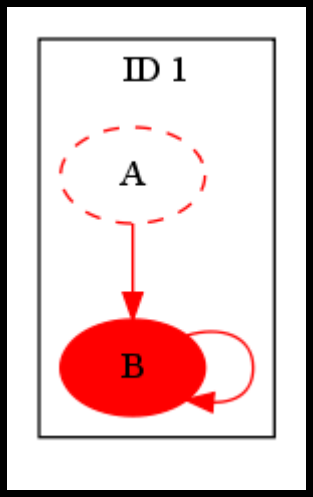
- Spezifikation von endlich vielen Zustandsautomaten

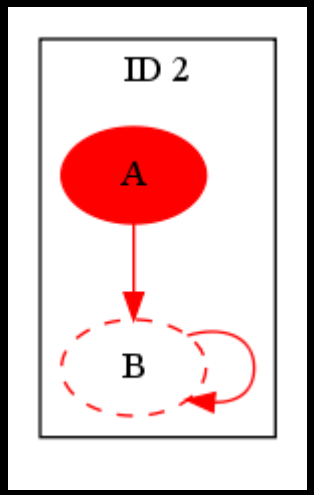


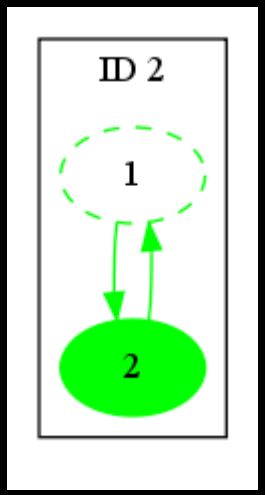


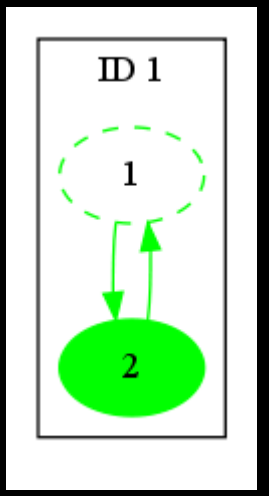
Paradigma

- Applikation ist eine Menge von Instanzen von Zustandsmaschinen







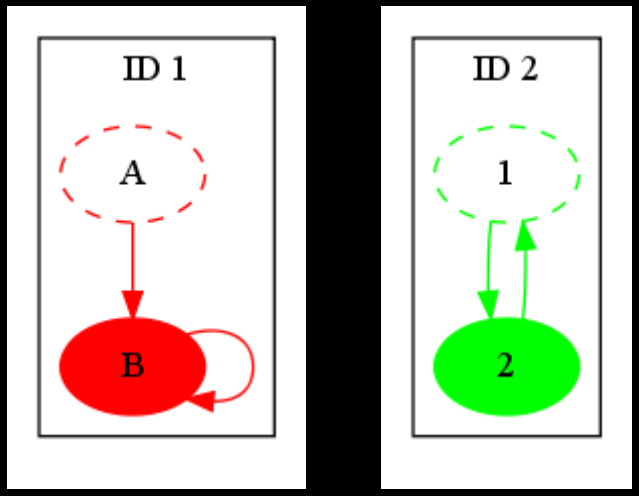


Partitionierung

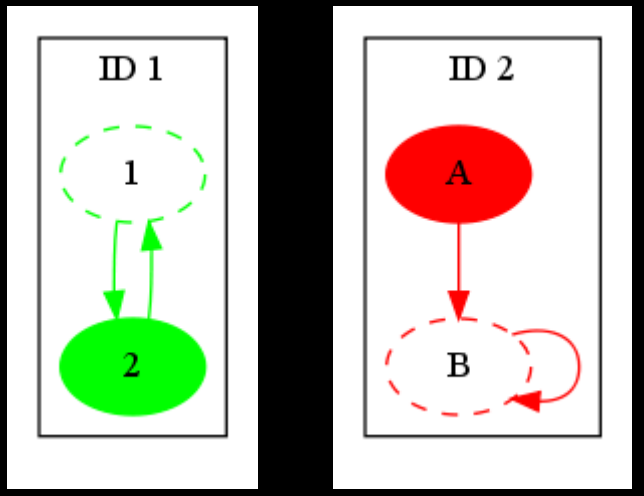
- pro CPU eine Partition
- pro Partition ein Worker-Thread
- statische Verteilung der Instanzen

Partitionierung

Partition 1



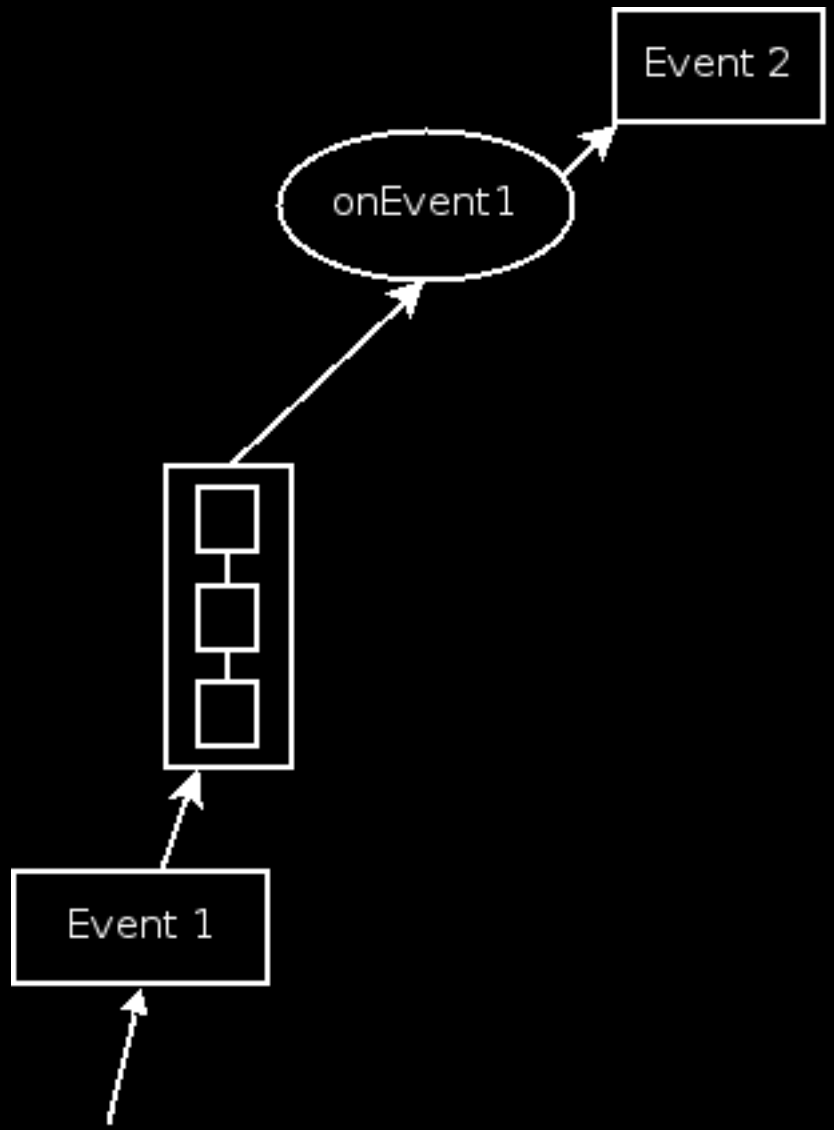
Partition 2



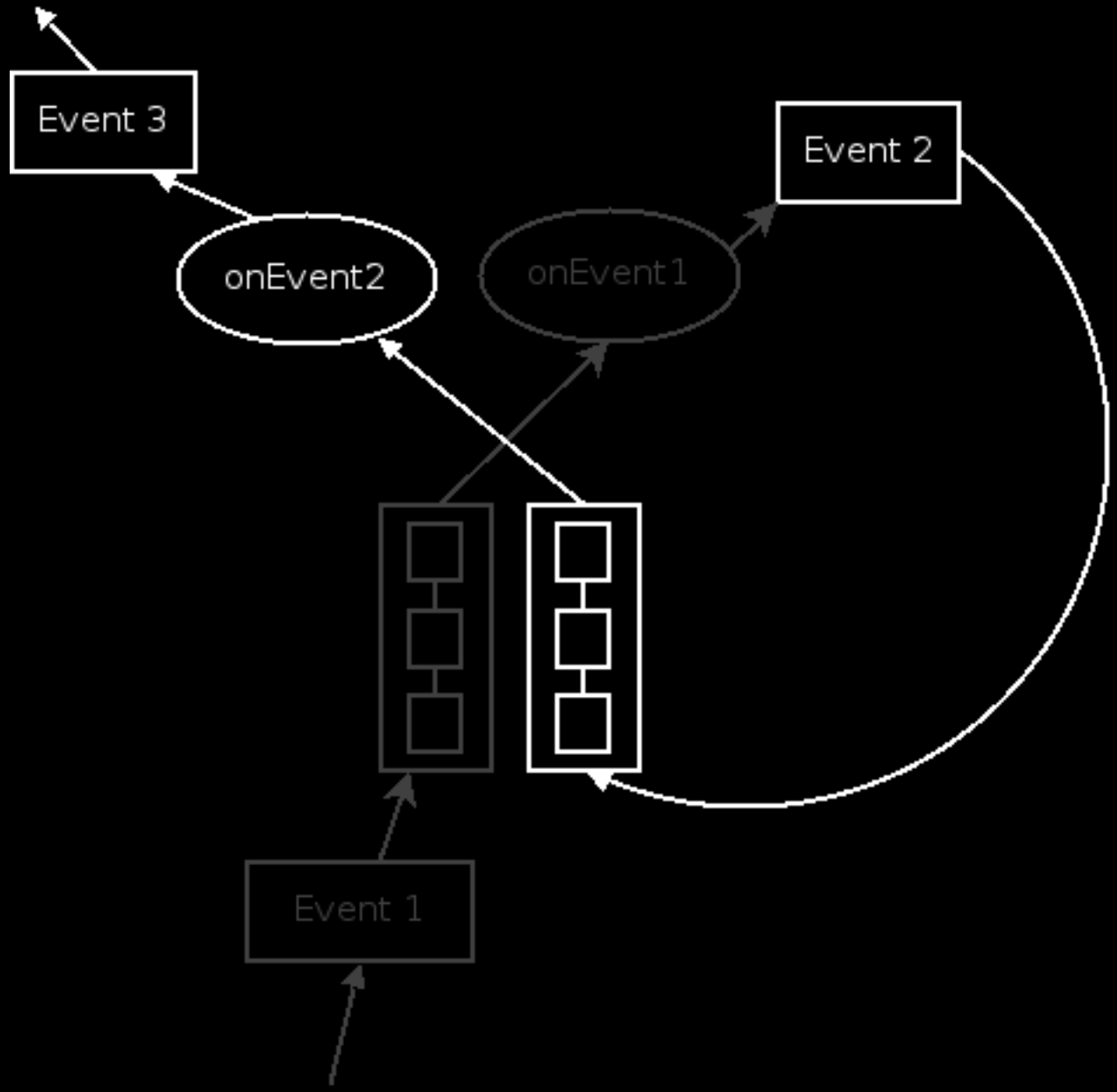
Übergänge

- Benutzerdefinierte Handler
  - werden von den Workern ausgeführt
  - müssen reentrant sein
  - dürfen nicht blockieren

Laufzeit



Laufzeit



Vorteile

- Threads werden nie unterbrochen
- sehr wenig Synchronisation
  - ein Leser – mehrere Schreiber
  - optimiert für Plattform
- kein manuelles Synchronisieren
- gleichmäßige, optimal Auslastung der CPUs

Nachteile

- Inversion of Control
- Bei wenig Last nicht optimal
- Sehr ungünstiger worst-case möglich
- Ausrichtung auf parallelisierbare Probleme
  - Partitionsübergreifende Zustände problematisch
- Probleme mit versteckter Synchronisation
  - Freispeicherverwaltung

Beispiel

primitiver Zugriffszähler

- Kommunikation über UDP
- Eingabe: Session ID
- Ausgabe: aktueller Zähler

Protokoll

```
$ netcat -u 127.0.0.1 9999
23
count is 1
23
count is 2
42
count is 1
23
count is 3
1
count is 1
```

Zustand

```
struct State {
   int counter;
};
```

Ereignis

```
struct Data {
  Toolbox::Sockaddr sender;
  std::string content;
};
```

Handler

```
struct Handler {
  void operator()(Data* data,
                  State* state)
  {
    state->counter++;
    // send reply to data->sender
  }
};
```

Schlüssel

- zu jedem Ereignis gehört ein Schlüssel
- Wird zur Partitionierung verwendet
- Jeder Schlüssel gehört zu genau einer Instanz

```
typedef uint8_t SID
```

Ereignis => Schlüssel

```
struct GetKey {
  SID operator()(Data* data)
  {
    return
      Toolbox::fromString<SID>
      (data->content);
  }
};
```

neue Instanzen

```
struct StateFactory {
    State* operator()(Data*)
    {
        return new State();
    }
};
```

Integration

- trotz C++ komfortable Notation
- Integration durch Typdeklarationen

Riot::Key

```
typedef Riot::Key<SID> RiotKey;
```

- Default ist Gleichverteilung

Riot::State


```
typedef Riot::State<
    State,
    RiotKey>
RiotState;
```

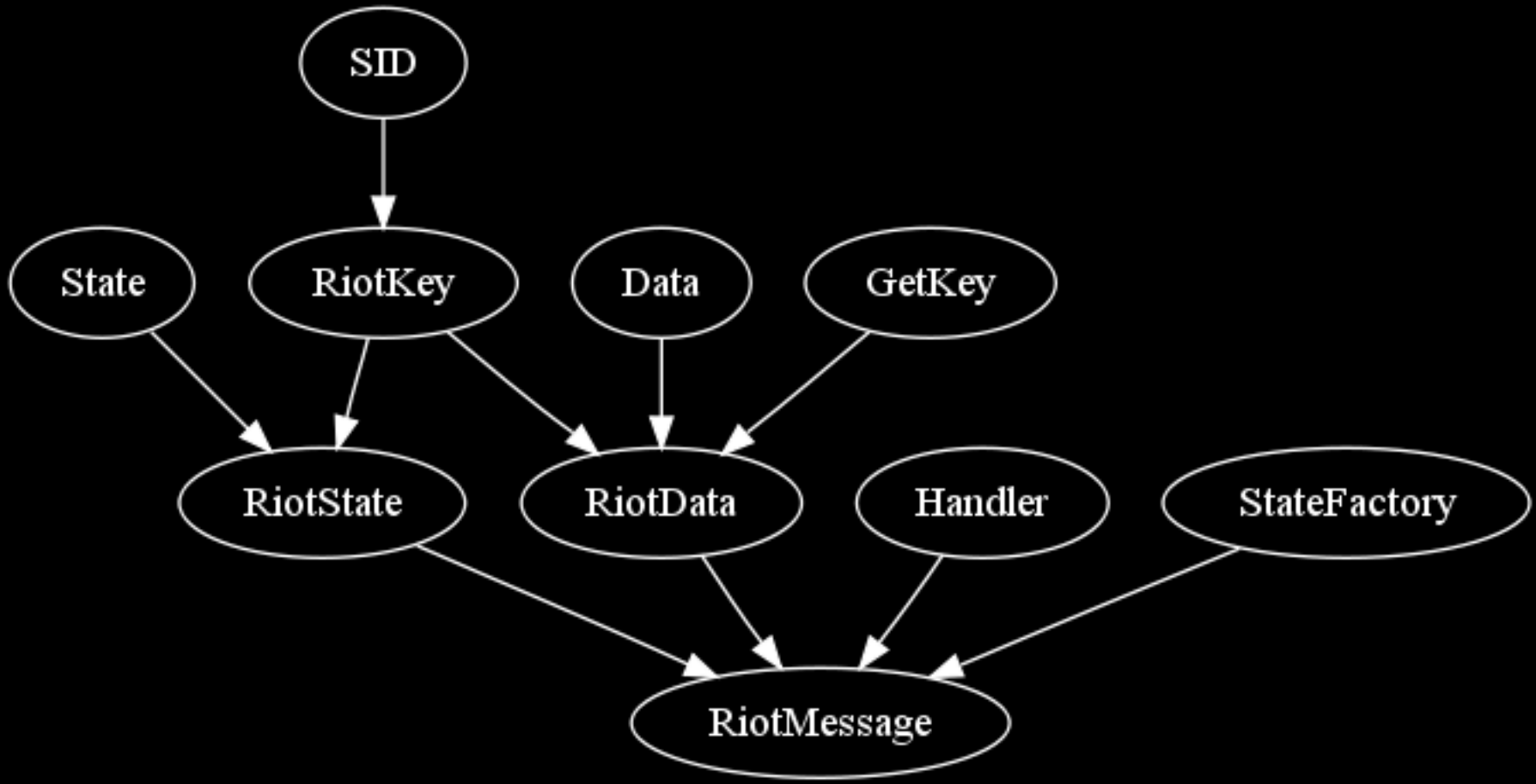
Riot::Data


```
typedef Riot::Data<
    Data,
    RiotKey,
    GetKey>
RiotData;
```

Riot::Message


```
typedef Riot::Message<
  RiotData,
  RiotState,
  Handler,
  StateFactory>
RiotMessage;
```

Überblick


SID
State
RiotKey
Data
GetKey
RiotState
RiotData
Handler
StateFactory
RiotMessage

Ereignis erzeugen

```
RiotMessage::emit(
     new Data(sender, content)
                    );
```

I/O

künstliche Zustände

- onEventA
  - action X
  - write
  - action Y
  - state = 2

- onEventA
  - action X
  - write_start
  - state = pending
- onWriteDone
  - action Y
  - state = 2

Implementierung

- sehr schlechter Support in POSIX
- eigene blockierende Threads
  - pro I/O-Resource ein Thread
  - Reservierung einer CPU für I/O Threads

Eingabe

- unabhängig von Riot
  - eigener Thread
  - blockierendes Lesen von einem Socket
  - Erzeugen von Riot Nachrichten

Ausgabe

- Verwendung von Riot
  - stellt Thread, Queue und Synchronisation bereit
  - spezielle, blockierende Partition
  - Handler schreibt blockierend auf einen Socket
- es gibt keine Zustände
  - eigene Nachrichten-Klasse
  - Verwendung der low-level API

Riot::Job


```
class Packet : public Riot::Job {
public:
  Result preDispatch();
  bool postDispatch();

private:
  Toolbox::Sockaddr saddr;
  const std::string data;
};
```

preDispatch

```
Result Packet::preDispatch()
{
  return
  Riot::Partition(
    UDP_SEND_PARTITION,
    true
                  );
}
```

postDispatch

```
bool Packet::postDispatch()
{
  socket.send(
    data.data(),
    data.size(),
    saddr
              );
  return true;
}
```

Ereignis absetzen

```
Riot::Scheduler::instance().
  addJob(
    new Packet(saddr, data)
        );
```

main

```
Riot::init(
   numberofPartitions,
   numberofBlockingPartitions
            );

Riot::start();
```

Features

Optionen

- `Riot::Key`
  - `GET_PARTITION`
  - `HASH`
  - `EQUAL`

Optionen

- `Riot::State`
  - `ADD_HANDLER`
  - `REMOVE_HANDLER`

Optionen

- `Riot::Data`
  - `GET_KEY`
  - `DESTRUCTOR`

Optionen

- `Riot::Message`
  - `STATE_FACTORY`
  - `AUTO_DESTRUCT`

Verwaltung von Zuständen

- `RiotState::addState`
- `RiotState::removeState`
- Auch über Partitionsgrenzen hinweg
- `RiotState::for_each`

Thread-local Storage

- Verwaltung von Zuständen der Partitionen

```
Riot::PartitionState<MyState>::get()
```

Initialisieren

- `Riot::start`
- `Riot::Initializer`
- `Riot::PartitionInitializer`
- `Riot::join`

Beenden

- `Riot::shutdown`
- `Riot::Job::destruct`
- `Riot::PartitionFinalizer`
- `Riot::Finalizer`

Timeouts

- Verzögertes Behandeln von Ereignissen
- manuelle Invalidieren nötig

Ausblick

Timeouts

- automatischen Invalidieren von verzögerten Events

Pinning von Threads

- alle Worker-Threads jeweils auf eine CPU
- alle I/O Worker gemeinsam auf eine CPU
- Plattformabhängig

Broadcast

- Nachrichten „an alle“
- mit oder ohne Synchronisation

Batch Jobs

- Absetzen von mehreren Ereignisen am Stück
- nur einmal Synchronisieren

Monitoring

- Statistik über die Auslastung
- Verbessern der Partitionierung

dynamische Partitionierung

- bei ungleicher Auslastung
- Umstellung der Partitionierung
- migrieren von States

QoS

- Priorisierung von Events

Riot in den Kernel

- asynchron ist schnell, synchron ist komfortabel
- Kernel und Applikationen sollten durchgängig asynchron arbeiten
- synchrone Abstraktion gehört in die Sprache

Verteiltes System

- eine Partition pro Server
- Event wird auf entfernten Server übertragen und dort behandelt
- einfacher Load Balancer

Danke

logix-tt

- Ulrich Dangel
  - Alpha- und Beta-Tester, Feature Requests
- Lars Rohwedder
  - Alpha- und Beta-Tester, Toolbox
- Rico Schiekel
  - Beta-Tester, Build System
- Volker Birk
  - initiale Anregung
  - produktiver Einsatz

Q&A