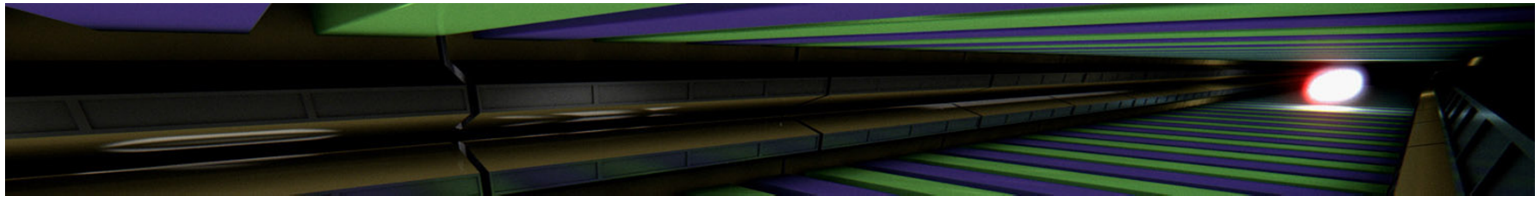


A homogeneous software framework

with *scientific computing* as an integral component

Burkhard Heisen for WP76
European XFEL GmbH User-Meeting
25 January 2012

What will be in this presentation?

- I am NOT talking about specific simulation or analysis software

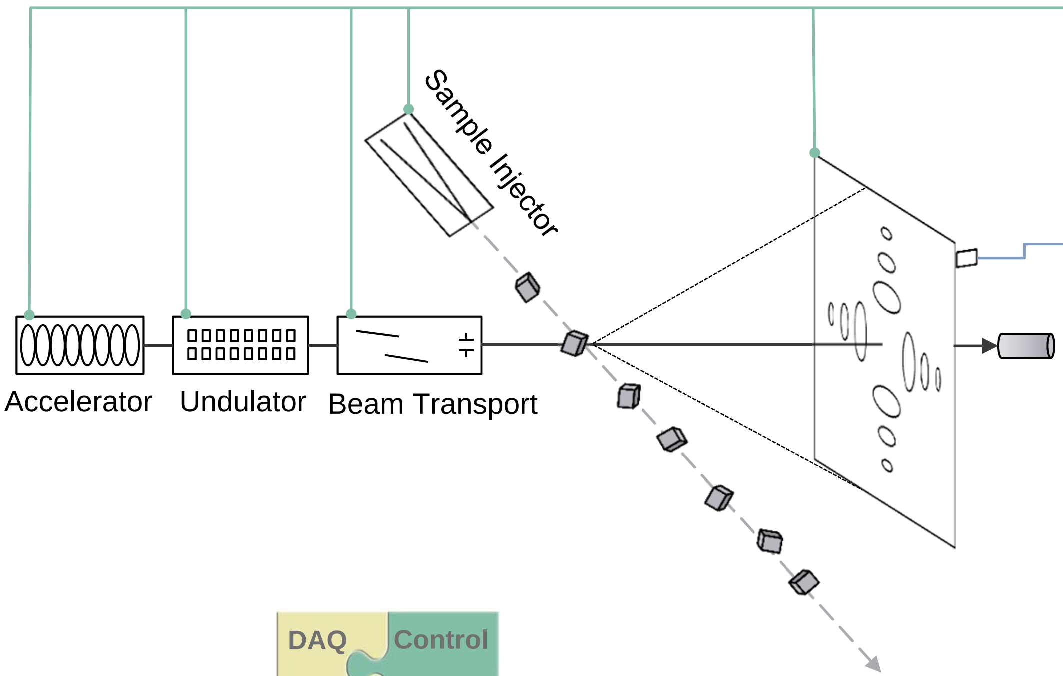
- What software is needed to enable users to run experiments?
  - Understand functional and technical requirements
  - A homogenous software framework is needed

- Conceptual ideas and initial implementation of the framework
  - Standardization and component re-usage
  - Managing distributed applications

- Scientific computing
  - Data pipelines
  - Image processing

What software do we need?

A typical use case:



Control
drive hardware and complex experiments
monitor variables & trigger alarms

DAQ
data readout
online processing
quality monitoring (vetoing)

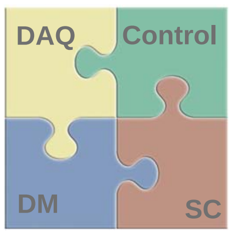
allow some control & show hardware status

show online data whilst running

setup computation & show scientific results

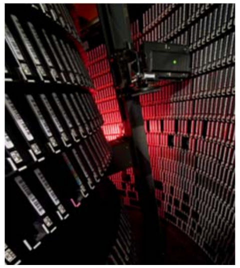
DM
storage of experiment & control data
data access, authentication authorization etc.

SC
processing pipelines
distributed and GPU computing
specific algorithms (e.g. reconstruction)



Tight integration of applications

XFEL.EU will need solutions for scientific computing



Experimental data is huge and must be stored local to XFEL.EU

No bulk data take home. We have to give users the possibility to analyze their data at XFEL.EU ("*data local computing*").

The huge amount of data needs special infrastructure to be efficiently processed

We have to give the users a simple way to make use of CPU/GPU cluster systems. Help understanding where data is, avoiding unnecessary duplication, keeping track of what has been done and when.

Beam time and storage is expensive, collecting useless data has to be avoided

Analysis whilst measuring is needed. Requires tight integration of DAQ, Control and SC.

Our own mandate – Technical requirements

- Enable **comm**... **ation** and fast data exchange **between applications** of any categ... ol, DAQ, Data Management, Scientific Computing)
- Provide a **unified inte**... equipment (hide details of hardware) and to all algorithms invo... torage or processing
- Make **integration**/development of **ne**... **nts** simple, intuitive and unambiguous
- **Hide the network**, be location transparent
- **Simple deployment and maintenance** including third-party resources

How can we achieve that?

Standardization of applications

- Proper standardization results in modular, scalable and homogeneous software

- It must however be guaranteed that all needed flavors of specialized applications can be developed within the standardized frame

- Before starting: check whether others have done something like that already
  - Most control systems standardize (e.g. Tango, Doocs) *software/hardware communication*
  - Big scientific packages do (e.g. CCP4, Phenix, Eman) *hkl handling, image processing*
  - Scientific workflow systems as well (e.g. Triana, Kepler) *data input/output, configuration*
  - However, we **found no system** that standardizes in such a "careful" way that our wide spectrum of functional requirements would be covered by a single solution
  - Composing different top-level software packages is difficult and leads to non-uniform software
  - **Decided to build the top-layer ourselves**, carefully learning from others and preparing to interface important systems

The homogenous software solution

- Identified components common to all software requirements

  memory/object management, configuration, logging, network services,
  error handling, data IO, python binding, databases, GUI,
  plug-in mechanism, cross-platform building and installation systems

- Decided to use C++ / Boost / Python / PyQt as core technology

- Do not re-invent the wheel, use high quality libraries under the hood

  Boost, Qt, OpenMQ, Log4cpp, TinyXML, Cimg, etc.

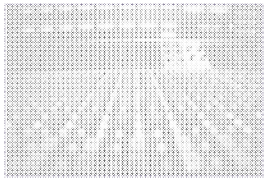
- Thought about concepts of how to deal with many distributed applications connected only via network

Main ingredients of a distributed system

Communication

Controller to Motor-Left: “Move 5 cm!”

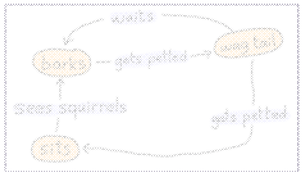
Compute-A to Compute-B: “I have an processed image available”



Configuration and Self-description

Motor-Right: “Hello, I am Motor-Right and my default velocity is 2 m/s.”

T1: “I am a PC-Layer device, will process exactly one train of frames.”

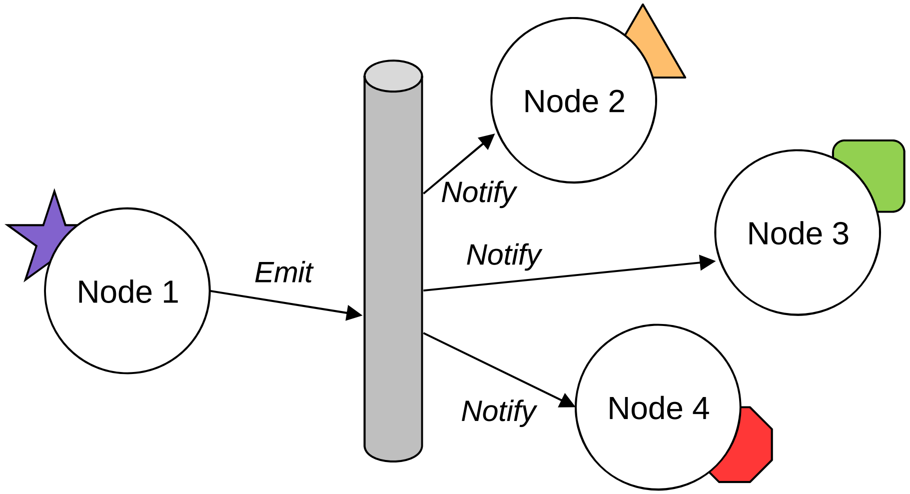


Flow-Control

Slit: “If Motor-Right also stops moving, I can report the new gap size.”

Compute-B: “Whilst I am processing, I can not read a new frame.”

Communication: Event-Driven vs. Scheduled



Event-driven communication "Push Model"

A minimal set of information is passed

System is scalable (maintains performance)

Failure is harder to detect

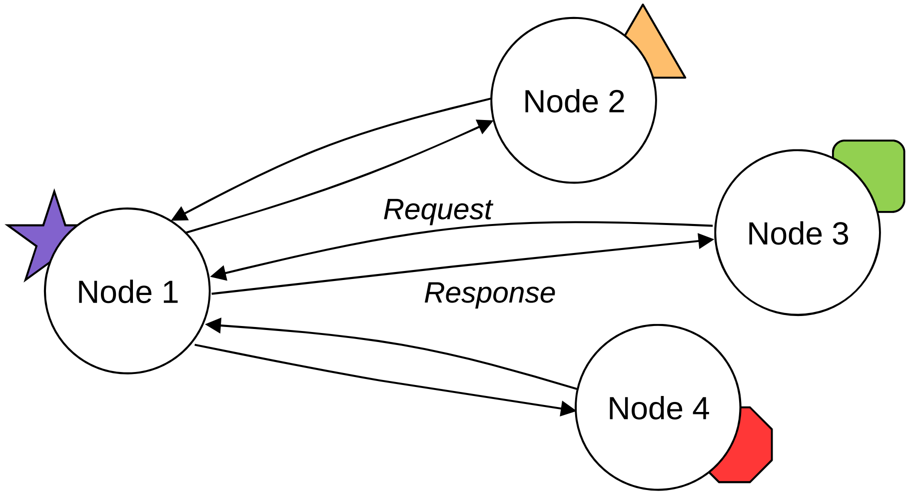
Scheduled communication "Poll Model"

Direct feedback on request

Nodes may be spammed (DOS)

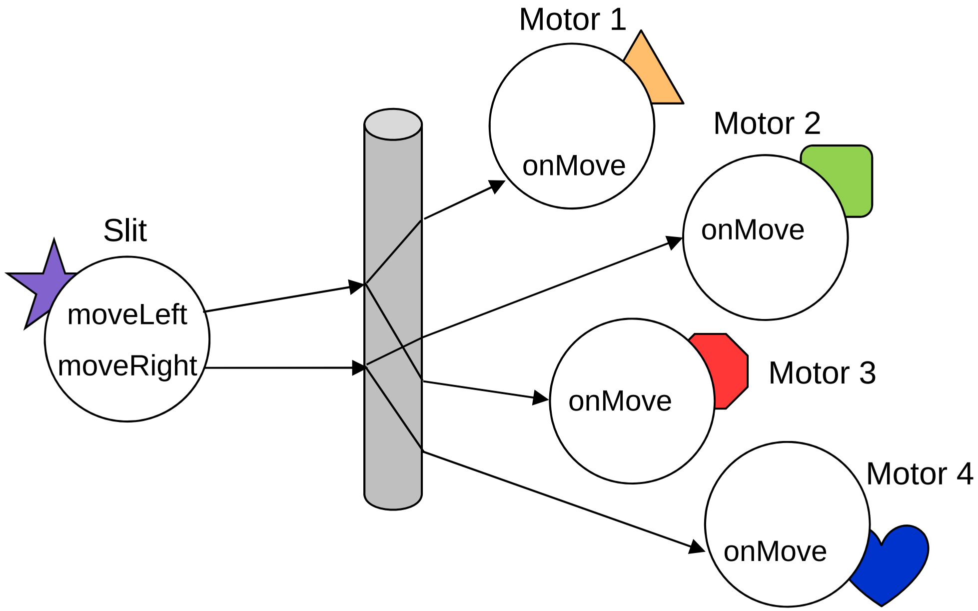
Growing systems loose performance

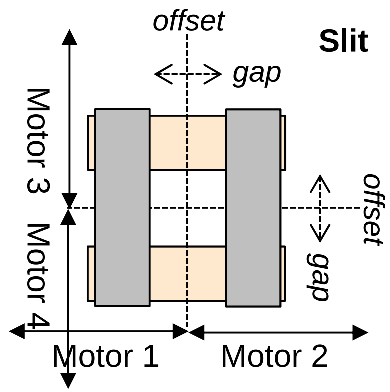
Typically, lots of extra traffic is generated



By the way… that is what Apple does:

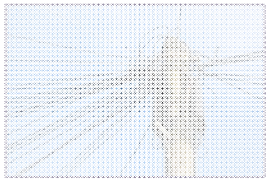
Communication API: Signals and Slots





- **Signal**: declares a command-name and the possible associated instructions
- **Slot**: declares a command-receiver and the possibly receivable instructions
- **Connect**: connects one signal of a specific source to one slot of a specific target
- **Emit**: executes a previously declared command with a specific instruction

Main ingredients of a distributed system



Communication

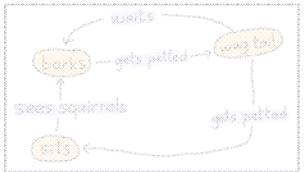
Controller to Motor-Left: “Move 5 cm!”

Compute-A to Compute-B: “I have an processed image available”

Configuration and **Self-description**

Motor-Right: “Hello, I am Motor-Right and my default velocity is 2 m/s.”

T1: “I am a PC-Layer device, will process exactly one train of frames.”



Flow-Control

Slit: “If Motor-Right also stops moving, I can report the new gap size.”

Compute-B: “Whilst I am processing, I can not read a new frame.”

Configuration and Self-Description

- Distinction between configuration at object construction and (re-)configuration of an existing object instance

- No need for user to validate any parameters. This is internally done taking the expectedParameters as white-list

- As the communication is also configurable, complex components can be composed using existing building blocks

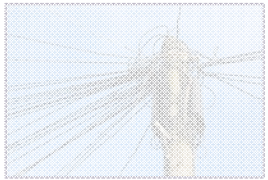
- Configurations can be converted to/from XML and XSD. Allows for a full-validated, full-controlled, strictly-typed plug & play architecture

Motor Device

```
expectedParameters {
  FLOAT_ELEMENT().key("velocity")
    .description("Velocity of the motor")
    .unitSymbol("m/s")
    .assignmentOptional().defaultValue(0.3)
    .maxInc(10)
    .minInc(0.01)
    .reconfigurable()
    .commit();

  INT32_ELEMENT().key("currentPosition")
    .description = "Current position of the motor"
    .readOnly()
    […]

  SLOT_ELEMENT().key("onMove")
    .description = "Trigger this slot to move the motor"
    .assignmentOptional().noDefault()
    .reconfigurable()
    […]
}

// Called once at initial construction
configure { […] }

// Called at each (re-)configuration request
onReconfigure { […] }
```

European XFEL

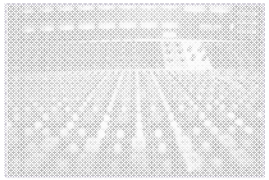
Main ingredients of a distributed system



Communication

Controller to Motor-Left: "Move 5 cm!"

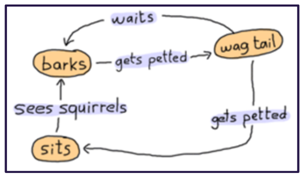
Compute-A to Compute-B: "I have an processed image available"



Configuration and Self-description

Motor-Right: "Hello, I am Motor-Right and my default velocity is 2 m/s."

T1: "I am a PC-Layer device, will process exactly one train of frames."

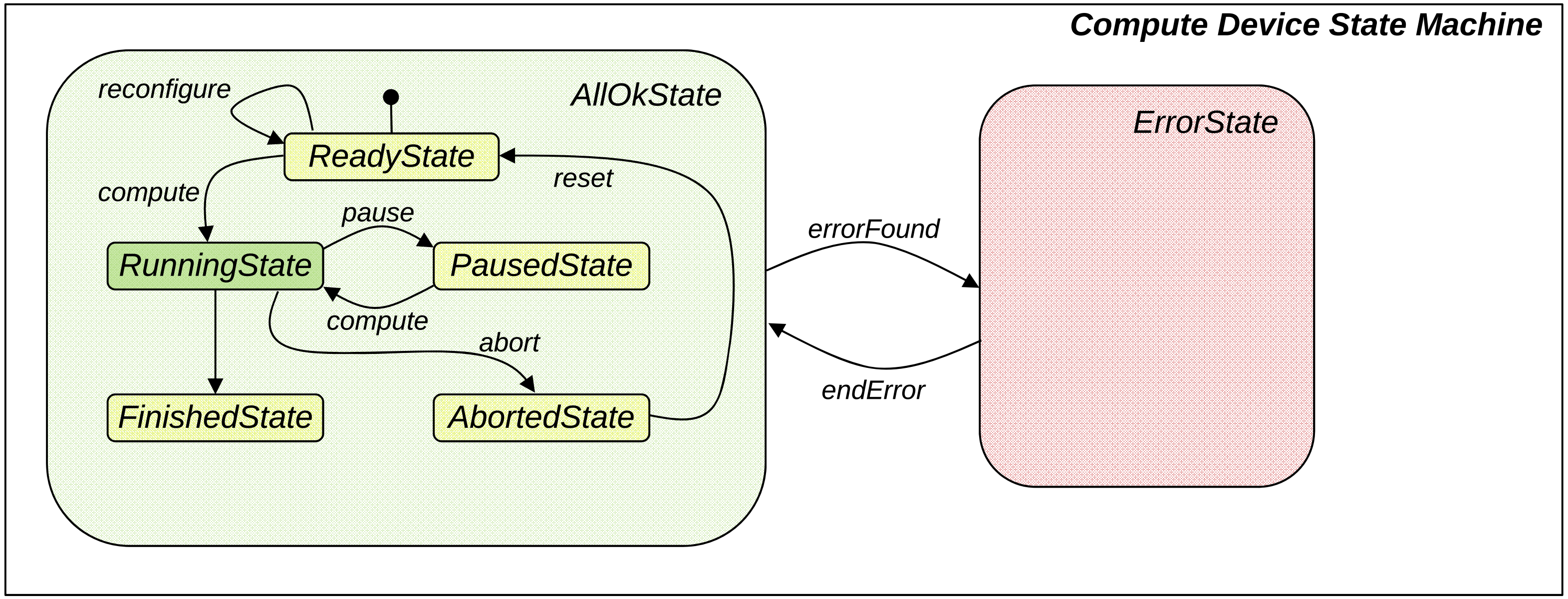


Flow-Control

Slit: "If Motor-Right also stops moving, I can report the new gap size."

Compute-B: "Whilst I am processing, I can not read a new frame."

Flow control – Using finite state machines



AllOkState

Source State	Event	Target State	Action	Guard
ReadyState	ReconfigureEvent	None	ReconfigureAction	IsValidReconfiguration
ReadyState	ComputeEvent	RunningState	ComputeAction	None
RunningState	PauseEvent	PausedState	PauseAction	None
RunningState	FinishedEvent	FinishedState	FinishAction	None

StateMachine

Source State	Event	Target State	Action	Guard
AllOkState	ErrorFoundEvent	ErrorState	ErrorFoundAction	none
ErrorState	EndErrorEvent	AllOkState	EndErrorAction	none

Main ingredients of a distributed system

Communication

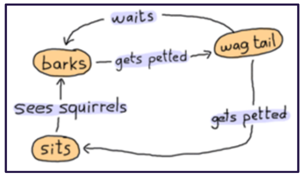
Controller to Motor-Left: "Move 5 cm!"

Compute-A to Compute-B: "I have an processed image available"

Configuration and **Self-description**

Motor-Right: "Hello, I am Motor-Right and my default velocity is 2 m/s."

T1: "I am a PC-Layer device, will process exactly one train of frames."

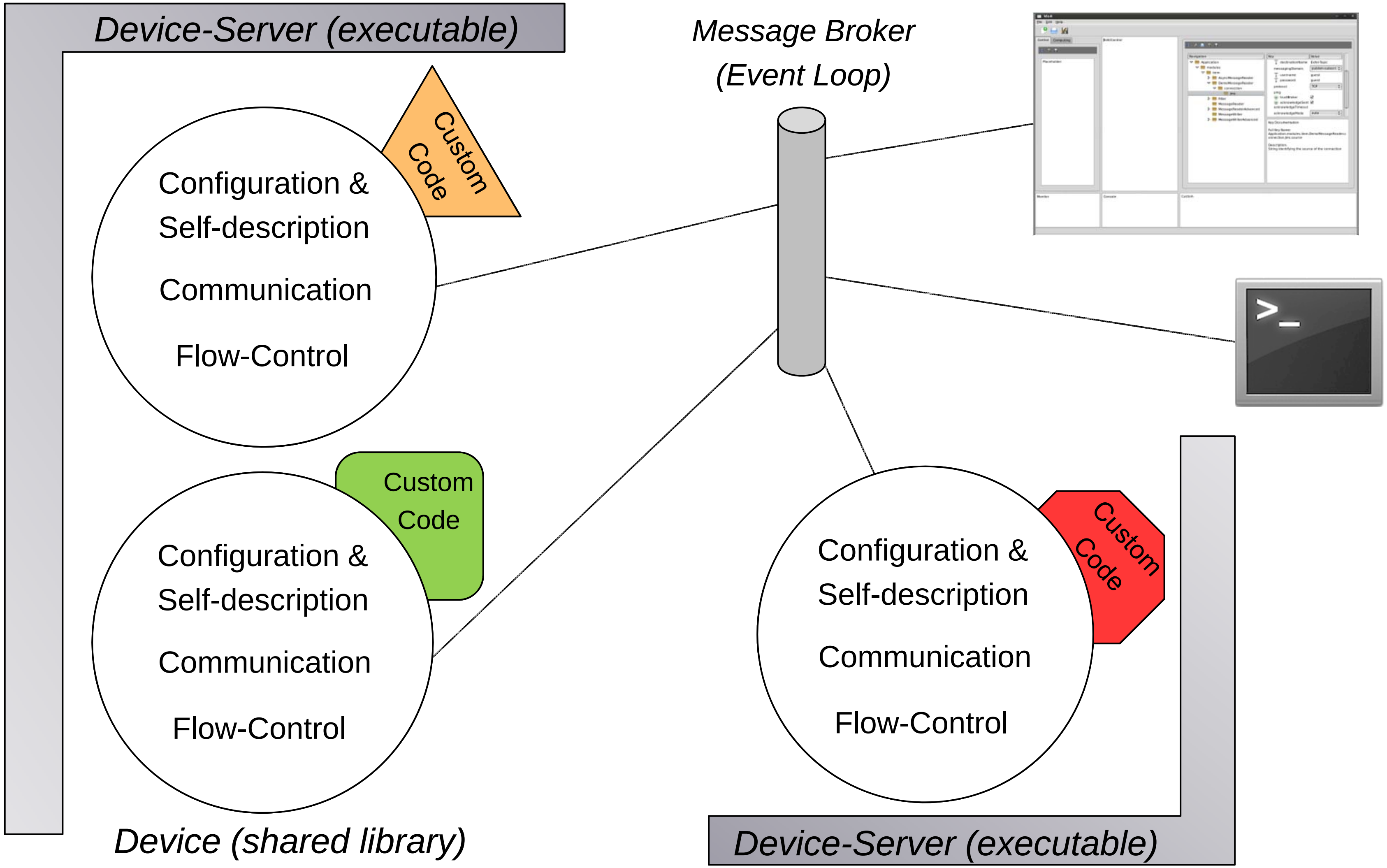


Flow-Control

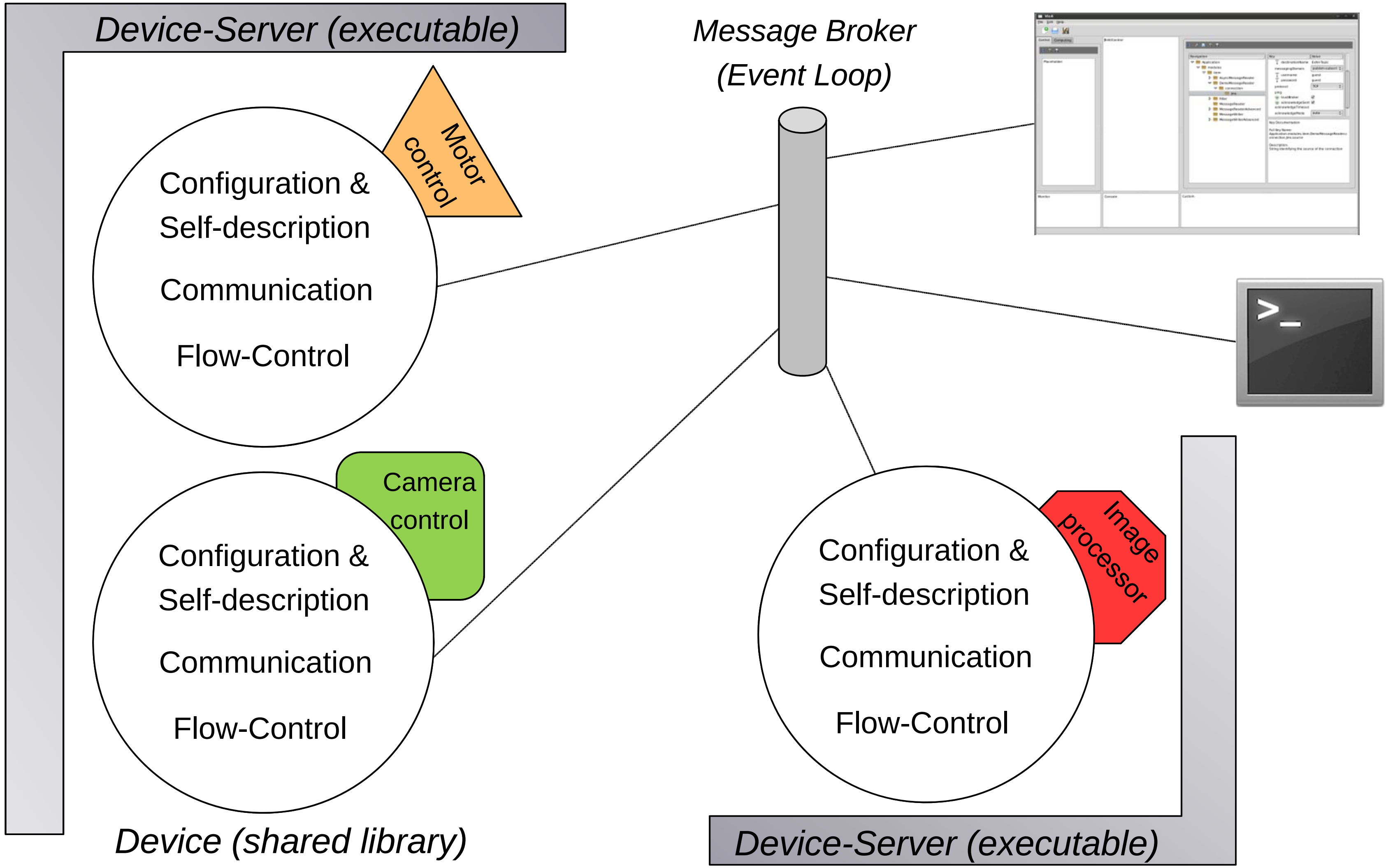
Slit: "If Motor-Right also stops moving, I can report the new gap size."

Compute-B: "Whilst I am processing, I can not read a new frame."

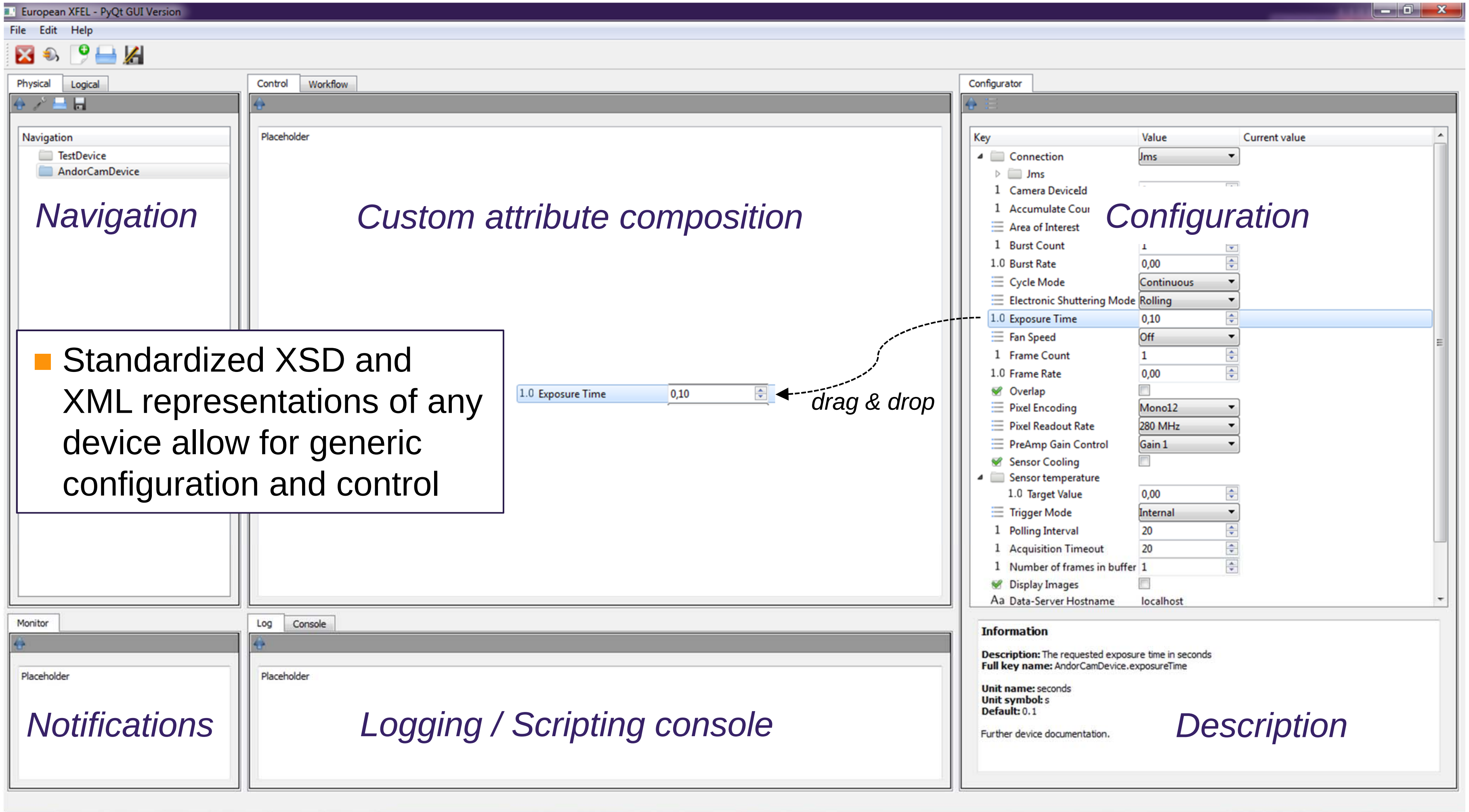
Putting it all together – Device Server & Devices



Putting it all together – Device Server & Devices



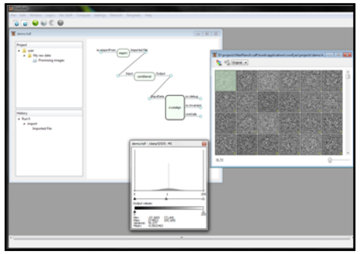
Multi-purpose GUI, thanks to standardization



- Standardized XSD and XML representations of any device allow for generic configuration and control

A bit more on scientific computing

Event driven data processing pipeline



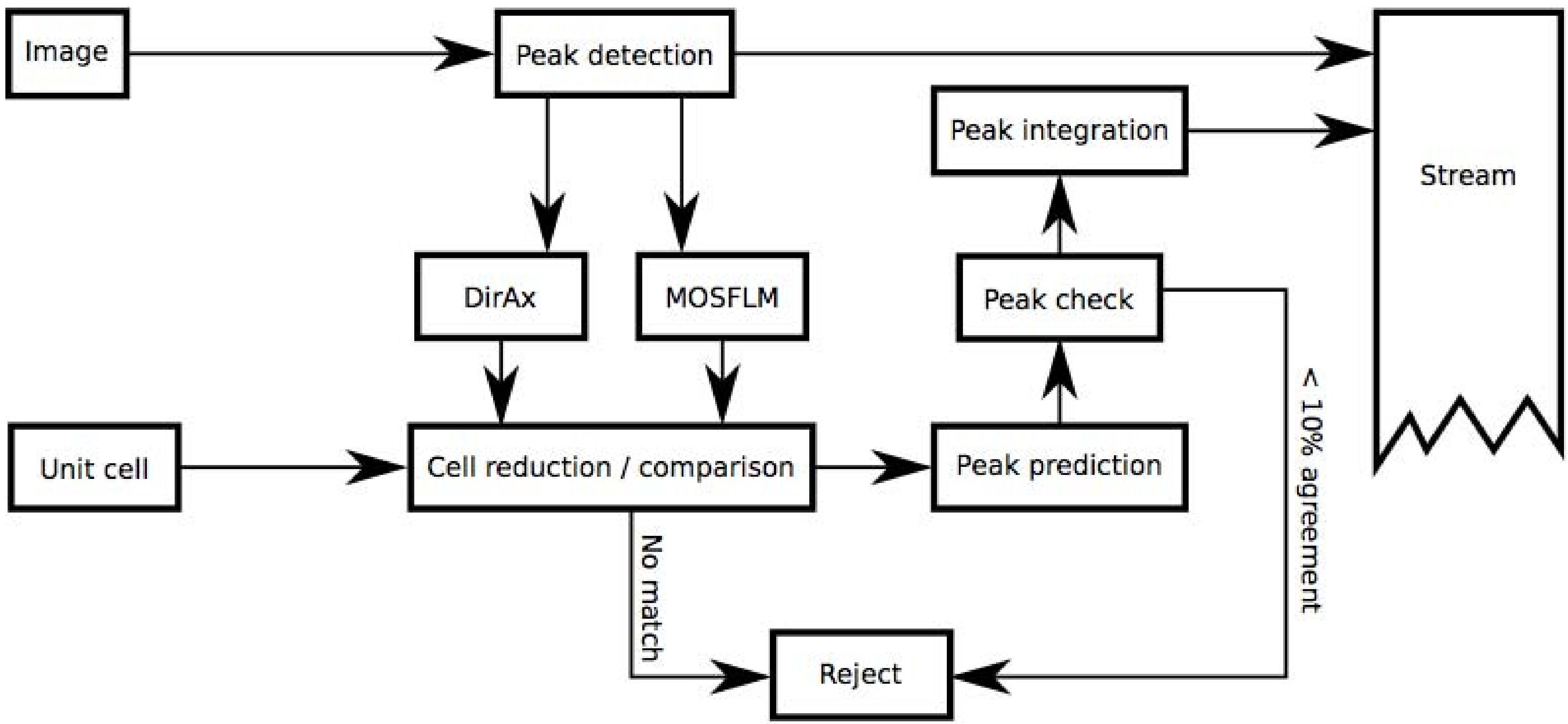
GUI components for data visualization and pipeline control

NVIDIA CUDA

CPU/GPU image processing utilities

Create workflows of modular components

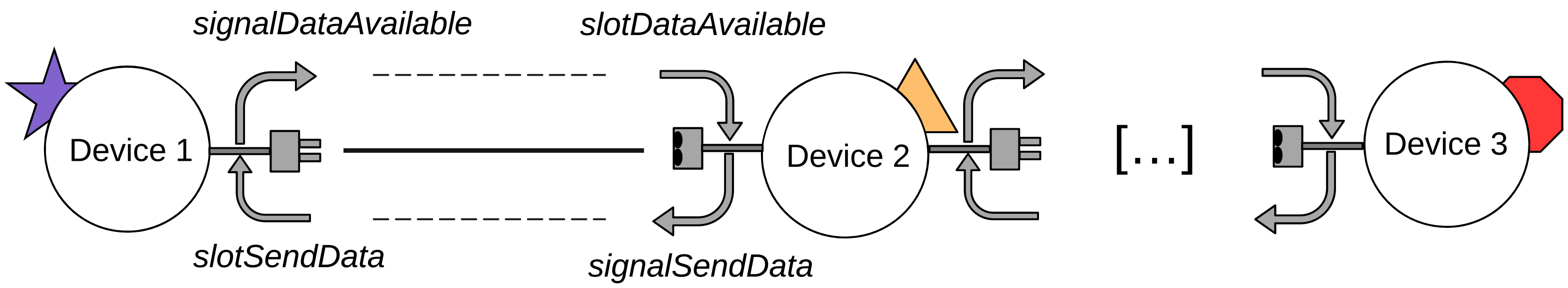
A conceptual SPB instrument workflow:



Taken from: BioFEL User Contribution

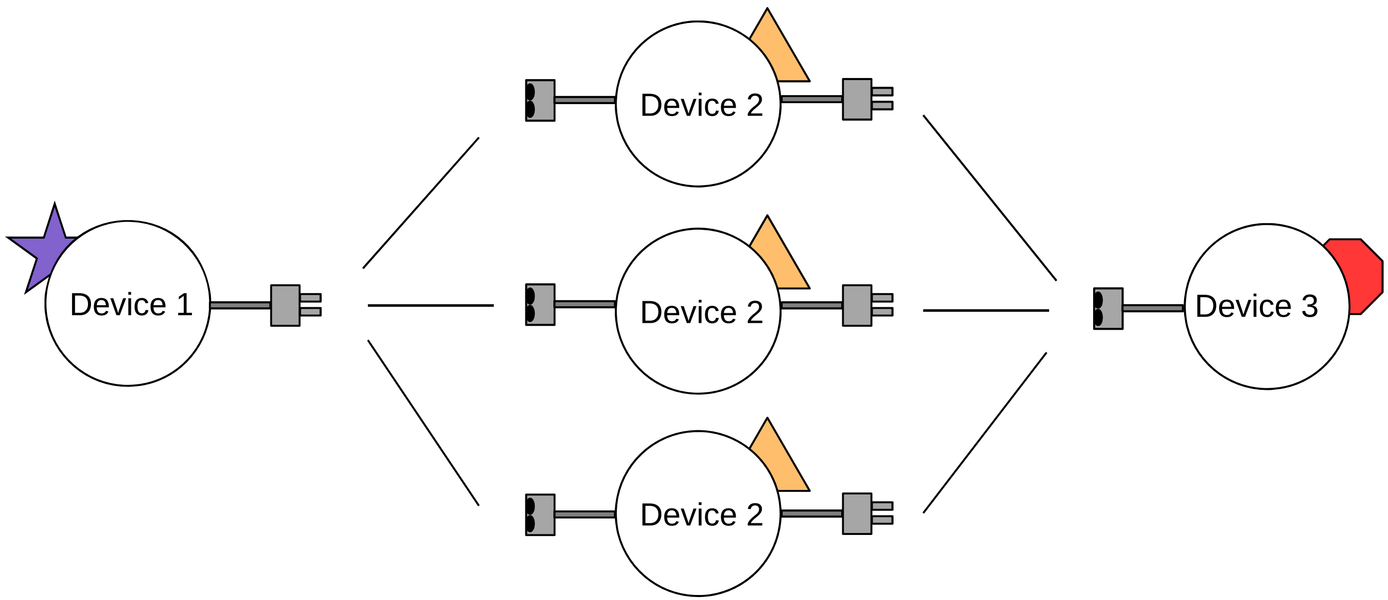
Pipelining of devices – Event driven data transport

- Data flow is controlled in an event driven manner



- Allow for flowing (streaming) data on a per image basis to minimize memory footprint
- Streaming modules can cache data on output channel
  - Provides failover if next module does not finish correctly
  - Provides fast re-execution of pipeline subsets
- Have possibility of collecting data for applications that need all data at once
- Have “adapter devices“ to integrate 3rd party applications “as is”

Parallelization as a design consequence



- Device level parallelization, thus transparent to developer
- Devices on same machine: CPU threads
- Devices on different machines: Distributed programming

A homogenous software framework with scientific computing as integral part

Pipeline system integrated in GUI

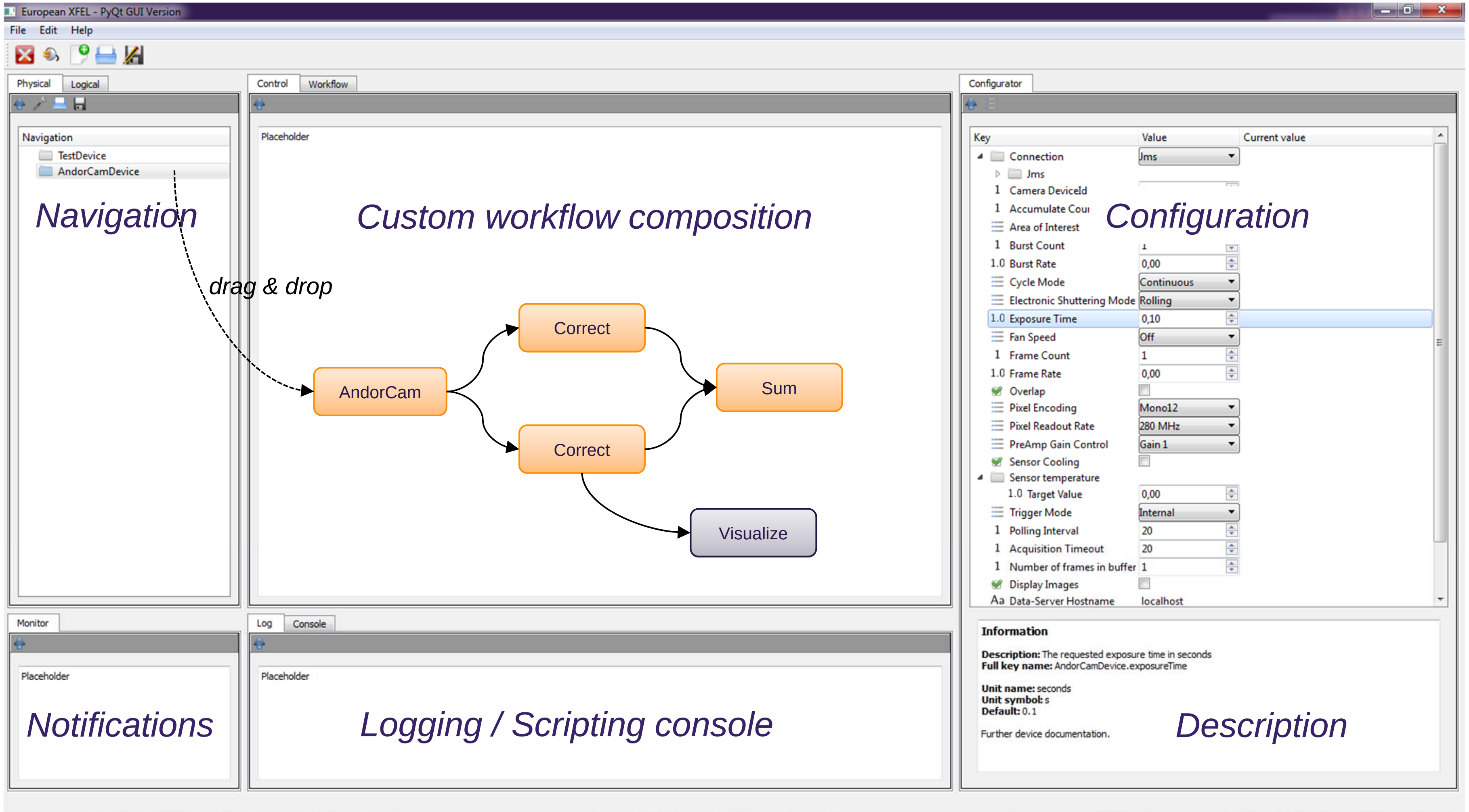
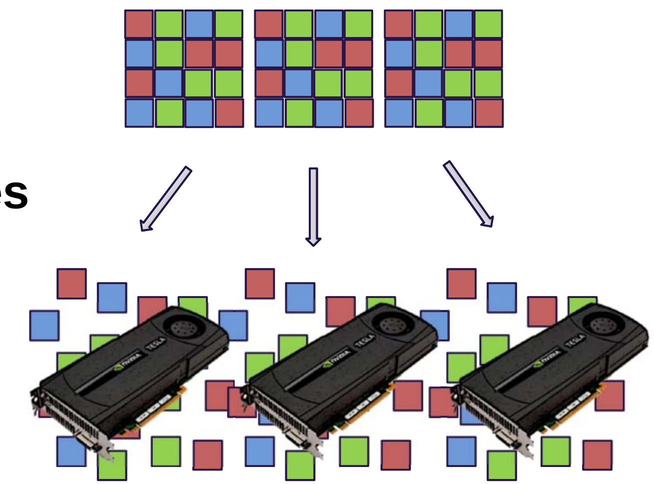


Image processing framework plans

- Integrate building process for **Nvidia CUDA** into the framework
- **Image classes** for both CPU and GPU
- Implementation of **standard processing routines**
- Provide **templates** for writing specific code on CPU or GPU
- Fully functional also **under Python**



"1 pixel per GPU thread"

What I have not talked about

- Load balancing, broker failover, network access restrictions
- Details of data management (privacy, aggregation, hdf5, etc.)
- User identification, role base locking systems (e.g. one controller at a time)
- Software packaging & installation, dependency maintenance, code&build@home
- Data provenance (i.e. record what has happened at each stage)
- Hardware synchronization requirements, TCP/IP vs. real-time systems

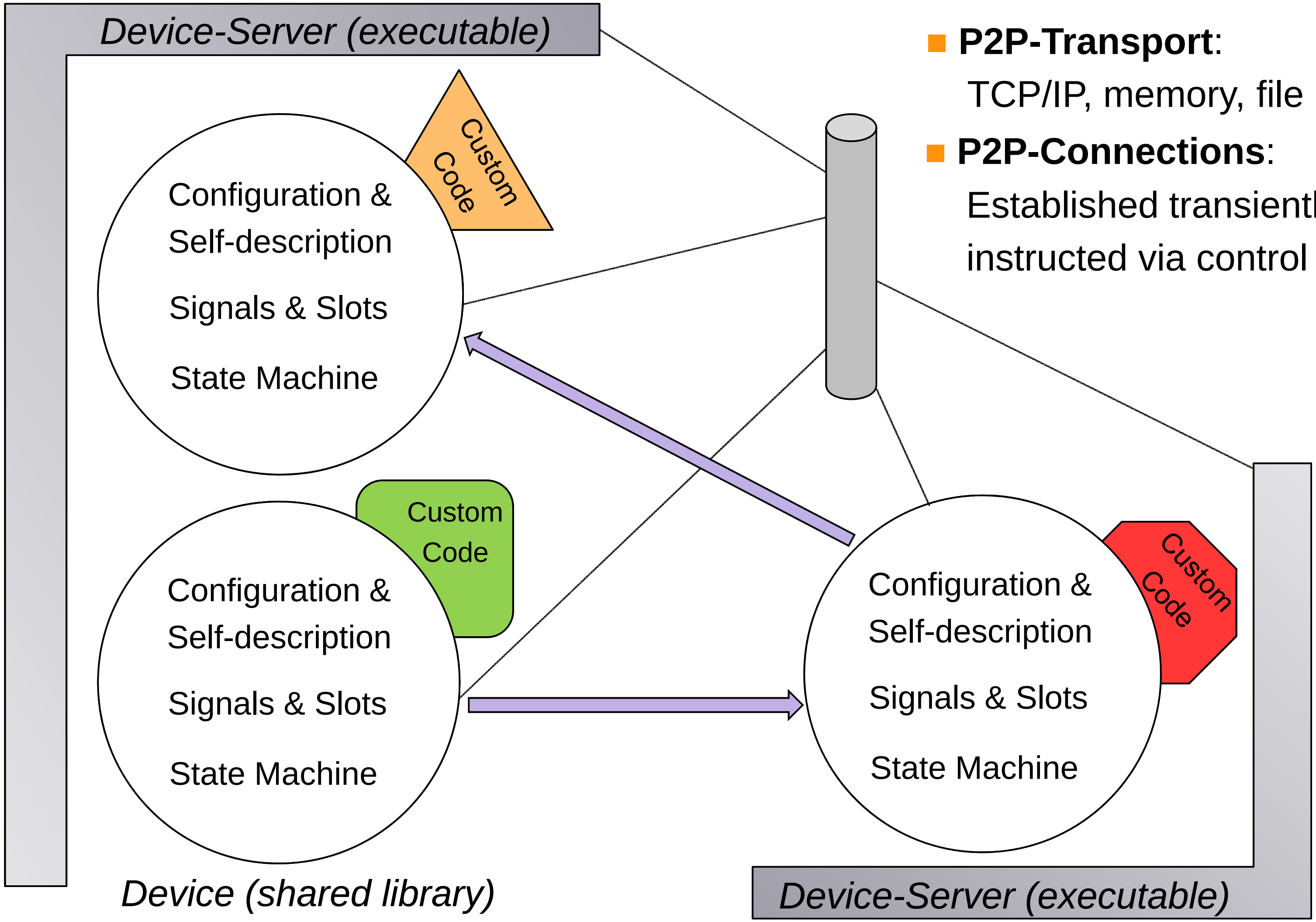
Conclusions

- XFEL.EU will provide **services for data storage** as well as **data analysis**
- The provided services focus on **solving general problems** like data-flow, configuration, project-tracking, logging, parallelization, visualization
- XFEL.EU software will be designed to allow **simple integration of existing algorithm/packages**
- It is the aim of XFEL.EU to **standardize the way data is stored and processed** amongst different experiments. This will allow an optimal usage of the available computing hardware infrastructure
- The ultimate goal is to provide a **homogenous software landscape to allow** fast and simple **crosstalk between all computing enabled categories** (Control, DAQ, Data Management and Scientific Computing)

Thank you for your kind attention.

Slides with more details

Pipelining devices – Large data flows point to point



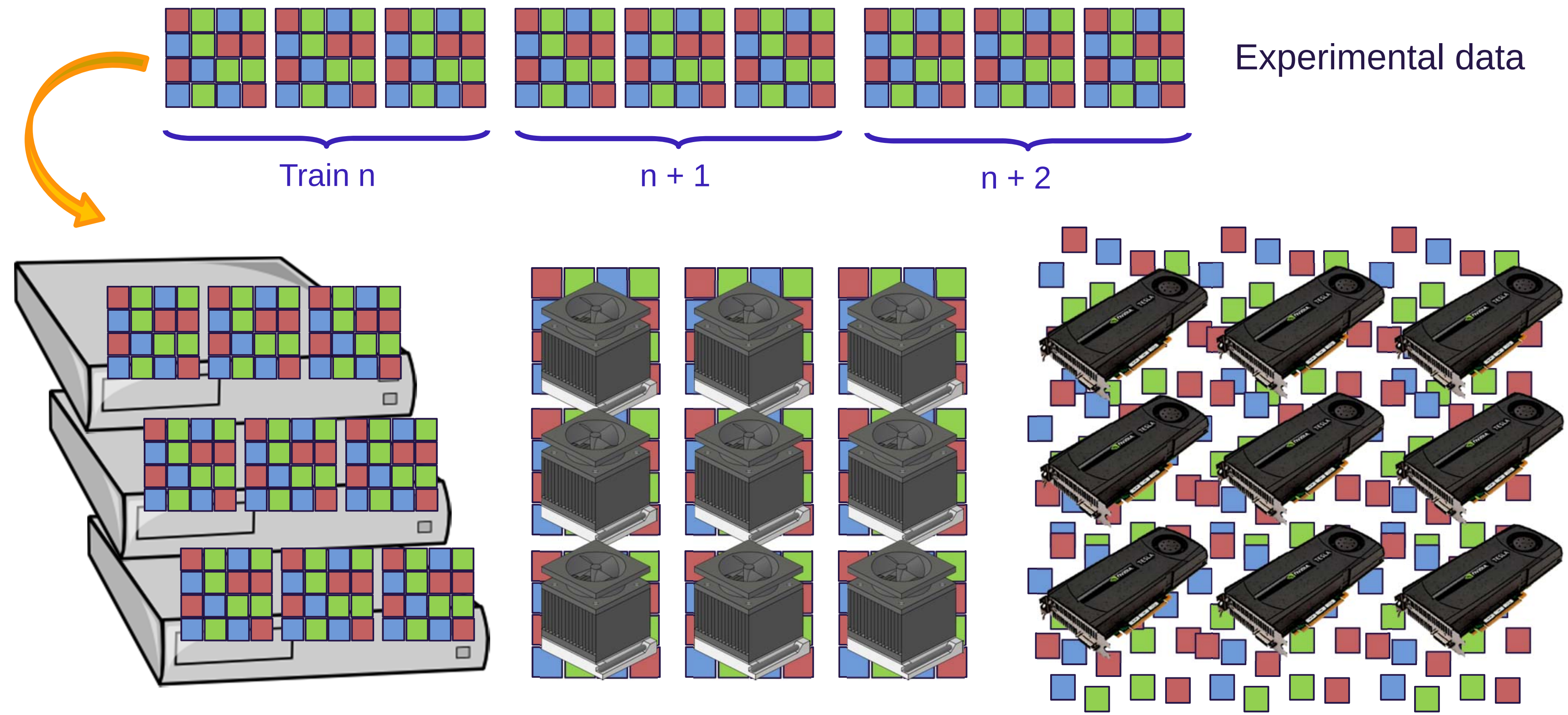
- **P2P-Transport**:
  TCP/IP, memory, file I/O
- **P2P-Connections**:
  Established transiently at runtime instructed via control information

Definitions

- What is a Device?
  - **Functionally**: A logical unit that is individually configurable and controllable. Can be regarded as a small application performing a specific task (e.g. steering a motor or filtering an image)
  - **Technically**: A (c++) class that inherits the device base class
  - **Architecturally**: A device is typically compiled into a shared library (.so/.dll)

- What is a Device-Server?
  - **Functionally**: An executable program that is able to run one or more devices
  - **Technically**: A (c++) class equipped with functions for parsing configurations(command-line, DB), loading plugins (devices), starting and stopping devices, etc.
  - **Architecturally**: A device-server is typically compiled into an executable (main)

Performance = parallelism = e.g. GPU usage

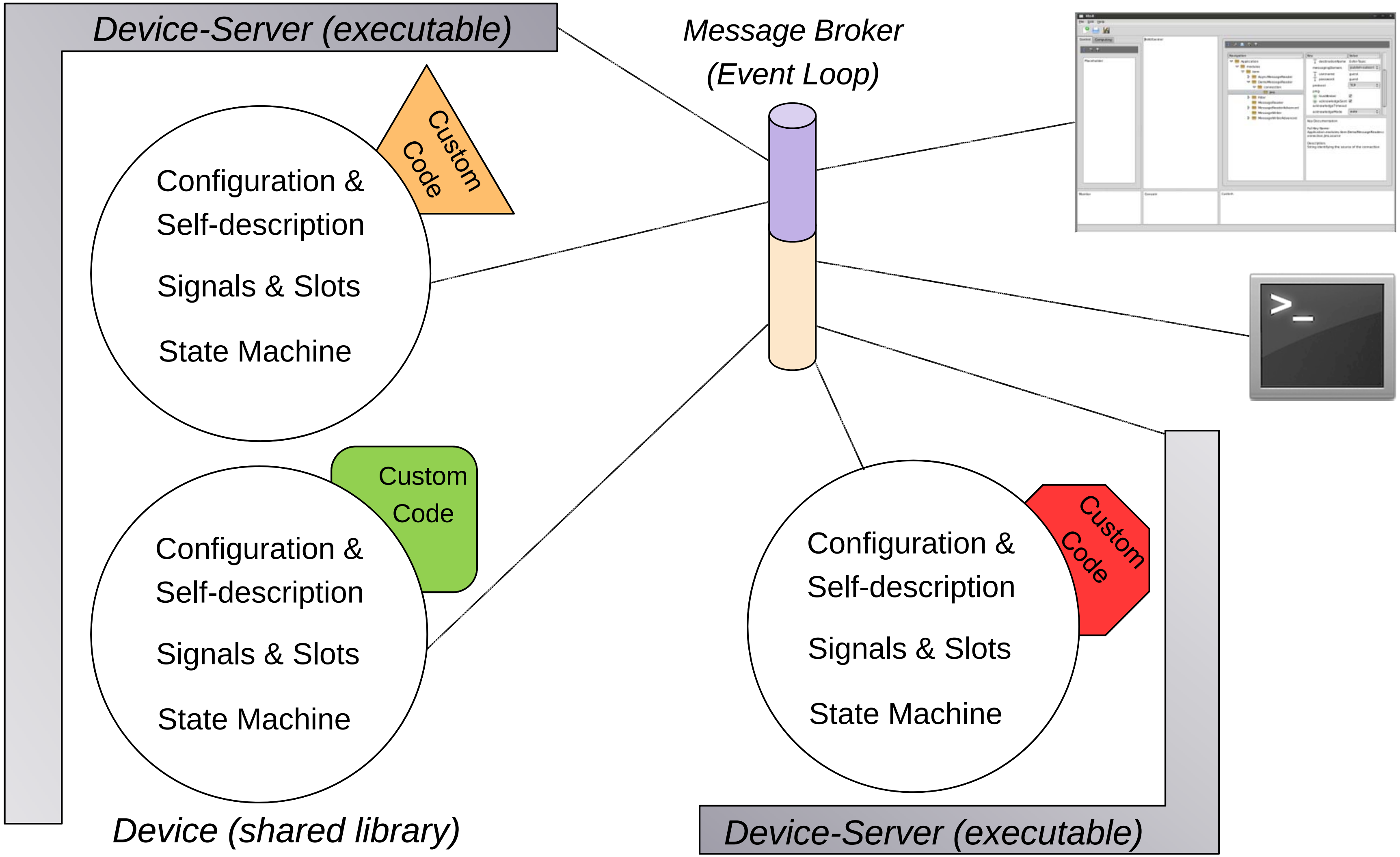


1. Level: Nodes (Computers)
"1 train per node"

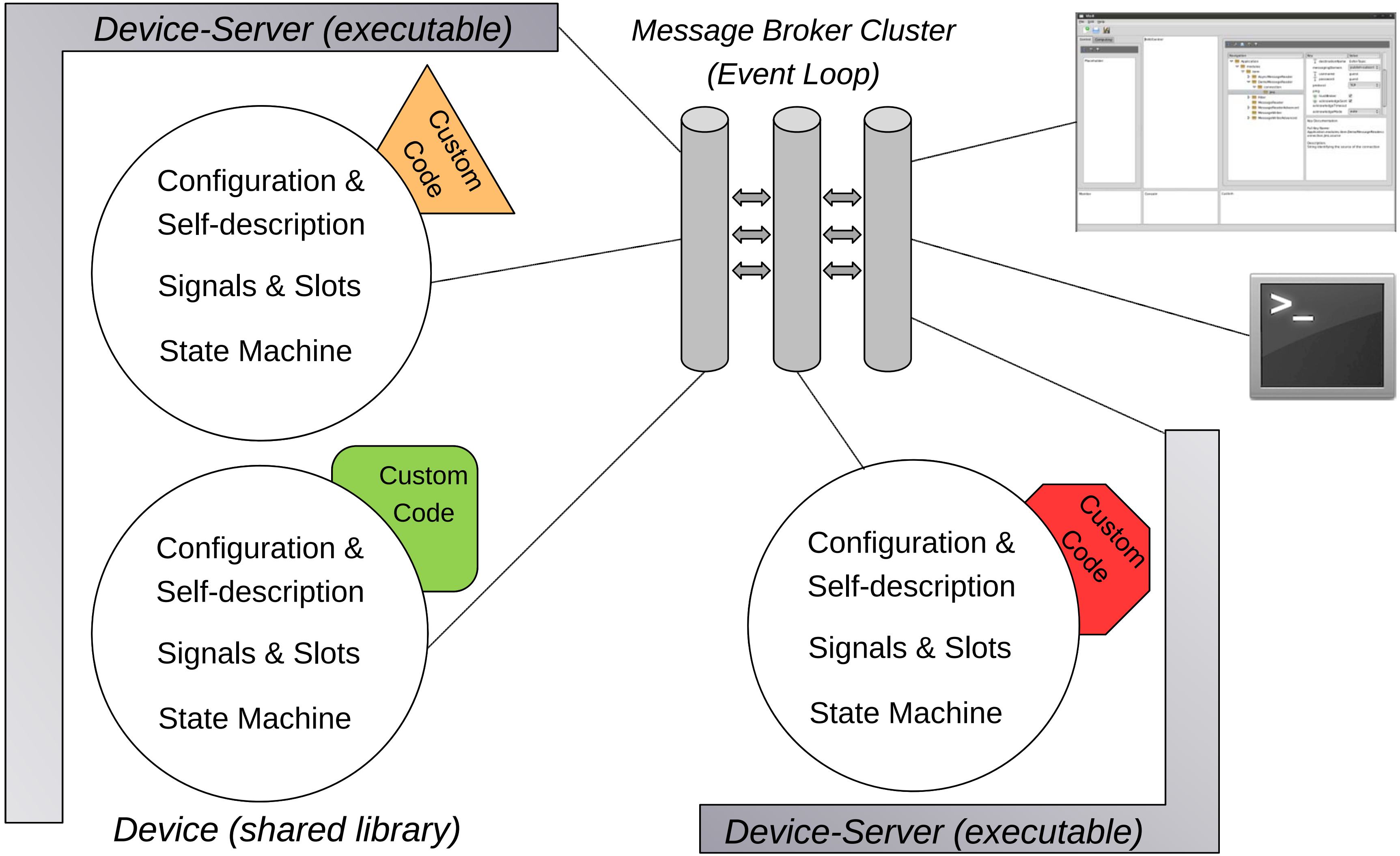
2. Level: CPUs
"1 frame per CPU thread"

3. Level: GPUs
"1 pixel per GPU thread"

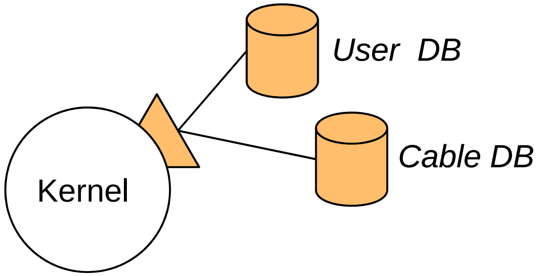
Splitting communication: Topics



Scale communication: Load balancing



Interesting devices: The kernel device



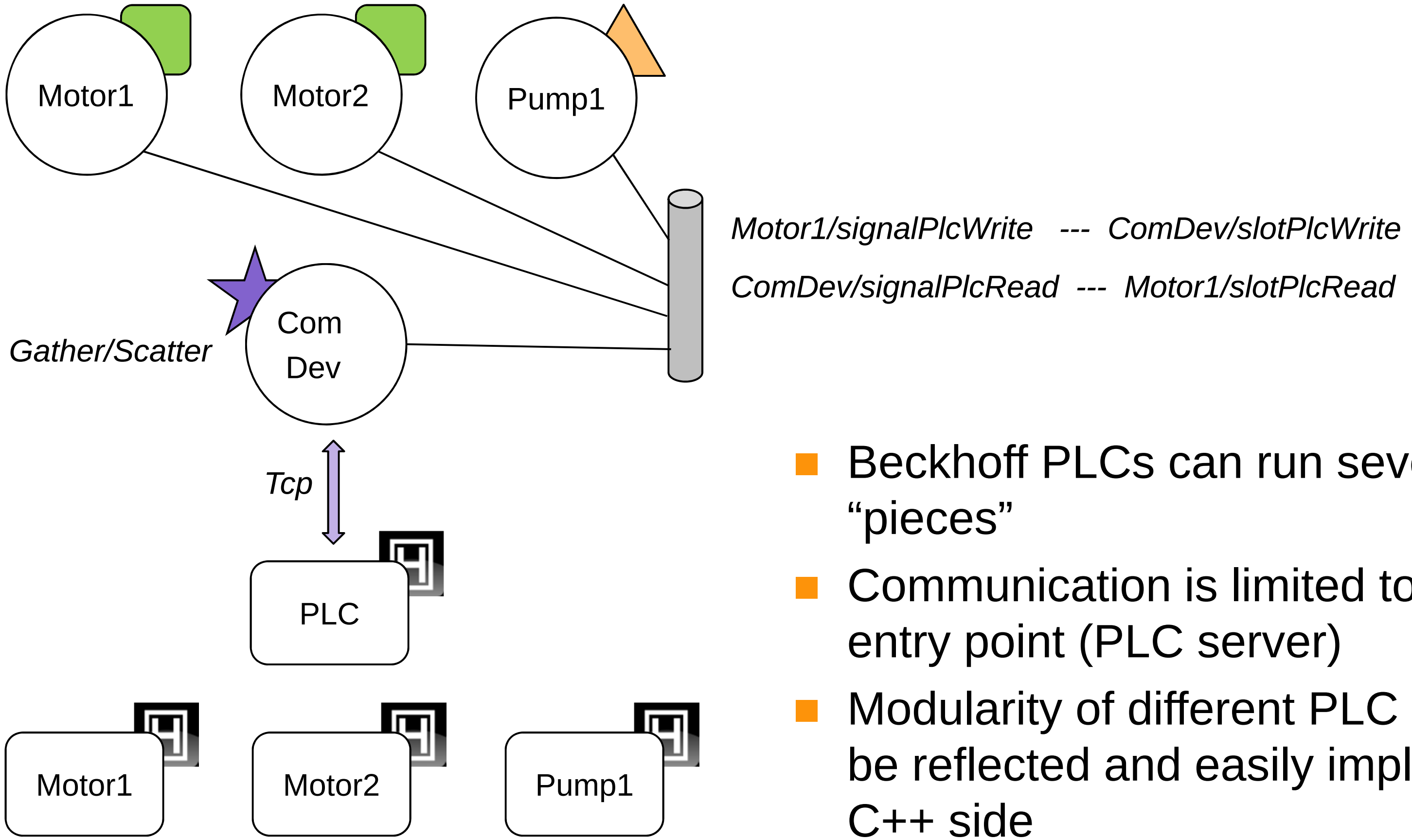
- **First device** to be started in the system
- **Connected to** one ore more **DBs** (user, cable, etc…)
- Serves as a **name server** for all other device-servers registering into the system
- Tracks all connects/disconnect requests:

  a) allows for user-based **access control on devices** (e.g. locking mechanisms)

  b) serves as watch-dog for **lost connections**, issues notifications/re-connects

  c) can be queried to provide selected connect information (e.g. for graphical displays)
- Knows the geographical location of each device-server (through cable DB)
- Keeps history about all information of the (control-)system
- May technically split into sub-devices for load balancing reasons

Interesting devices: The BeckhoffCom Device



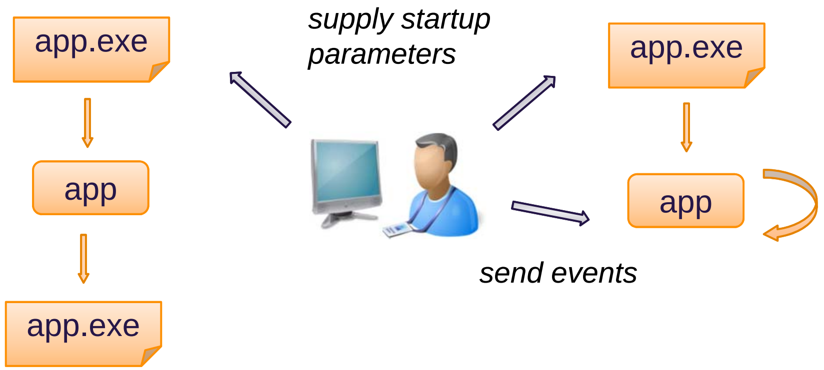
- Beckhoff PLCs can run several hardware “pieces”
- Communication is limited to a single entry point (PLC server)
- Modularity of different PLC setups should be reflected and easily implemented on C++ side

Applications

Lifecycle:

Procedural

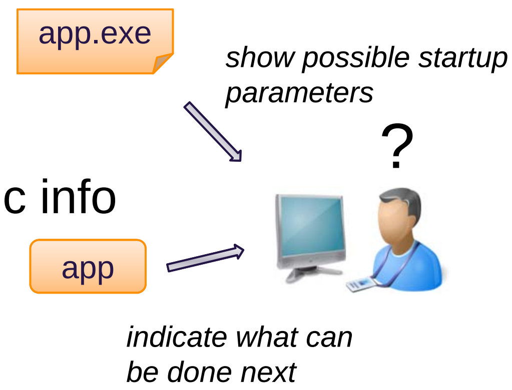
Interactive



How to use:

Static info

Dynamic info



The Device – A standardized application

static **expectedParameters:**

- Developer defines needed/available **attributes** (input/output channels, program parameters)
- He decides when attributes can be used (startup only, interactively*) and how (read/write flags)
- For each attribute/command the developer adds as much **additional description** as possible

static **programFlow*:**

- Developer defines how the application can behave if used interactively
- He defines **states**, **events, actions,** and a **flow-table** showing what happens when

configure:

- This function is called only once at startup
- Provides (validated) access to all above described attributes

run:

- This function is called once after configure
- Procedural: Write any code and it will execute here
- Interactive: Start the programFlow which blocks the application here, custom code must be written above defined state entries/exits or actions

onStateA_Entry (onStateA_Exit**):**

- Hook as defined in programFlow

onActionX:**

- Hook as defined in programFlow

Current status of the toolkit

Factories	Type introspection	Configurations	Logging
Plugins	Network services	Messaging	Signal/slot
IO interfaces	FSM	Python integration	GUI
Databases	Image processing		

Not implemented | Implemented

Our extension: Cross-network Signals & Slots

Technical realization	Qt	CNSS

Qt

```
class Motor : public QObject {

  Q_OBJECT

signals:
  void move(int);

public slots:
  void onMove(int);
};
```

CNSS

```
class Motor : public SignalSlotable {

  Motor() {
    SIGNAL1("move", int)
    SLOT1(onMove, int)
  }

  void onMove(const int&);

};
```

Our extension: Cross-network Signals & Slots

Technical realization	Qt	CNSS

Qt

```
That* that = new That();

//       src          signal          tgt           slot
connect(this, SIGNAL(move(int)), that, SLOT(onMove(int)));
```

CNSS

```
//       src     signal        tgt           slot
connect(“”, “move-INT32”, “ds1/m1”, “onMove-INT32”);

// Or:      signal                  slot
connect(“move-INT32”, “ds1/m1/onMove-INT32”);
```

Our extension: Cross-network Signals & Slots


```
// Blocks until slot execution
emit move(7);
```

Qt

```
// Immediately returns
emit("move", 7);
```

CNSS

Technical realization	Qt	CNSS
Connection	limited to object instances (pointers)	on different applications/platforms (hostId/instanceId)
Emit	Typically blocks, multiple slots are called sequentially (synchronous & event-driven)	Never blocks, multiple slots are called concurrently (asynchronous & event-driven)

Our extension: Cross-network Signals & Slots

Technical realization	Qt	CNSS
Declaration of Signals/Slots	Before compile time (moc-tool), no static or global slots	At runtime, static and global slots are possible
Connection	Source and Target are limited to object instances (pointers)	Source and Target can be on different applications/platforms (hostId/instanceId)
Emit	Typically blocks, multiple slots are called sequentially (synchronous & event-driven)	Never blocks, multiple slots are called concurrently (asynchronous & event-driven)
Event propagation	Direct function calls (FIFO array of function pointers)	Events are MOM messages (message-queue servers as event stack)

Flow-Control – Again, reusing a very successful concept

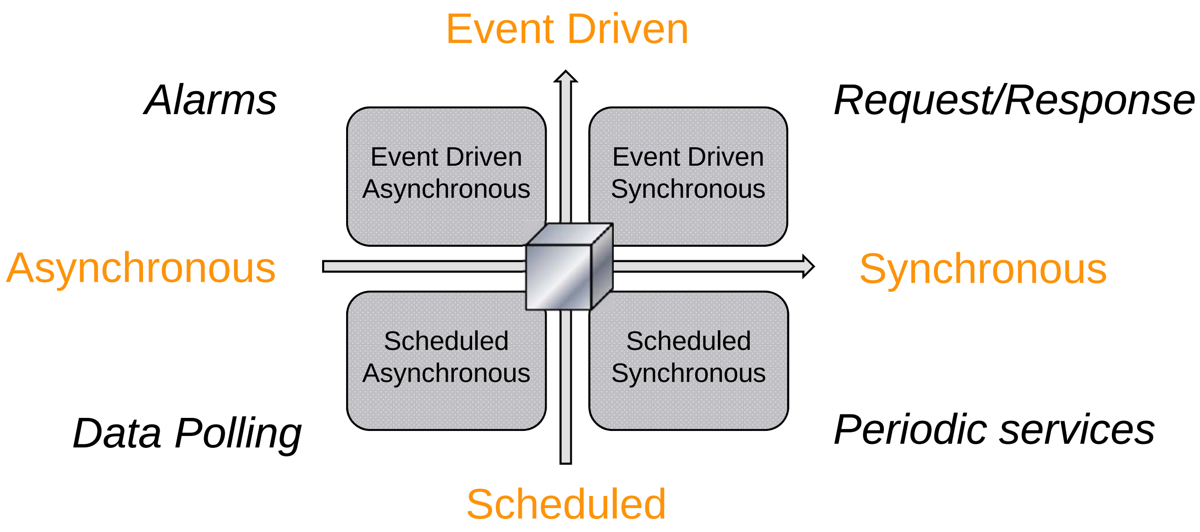
Structure program flow using Boost's: MSM (meta-state-machine)

- **State Machine**: the life cycle of a thing. It is made of states, transitions and processes incoming events.
- **State**: a stage in the life cycle of a state machine. A state (like a submachine) can have an entry and exit behaviors
- **Event**: an incident provoking (or not) a reaction of the state machine
- **Transition:** a specification of how a state machine reacts to an event. It specifies a source state, the event triggering the transition, the target state (which will become the newly active state if the transition is triggered), guard and actions
- **Action:** an operation executed during the triggering of the transition
- **Guard:** a boolean operation being able to prevent the triggering of a transition which would otherwise fire
- **Transition Table:** representation of a state machine. A state machine diagram is a graphical, but incomplete representation of the same model. A transition table, on the other hand, is a complete representation

Advantages of thinking in Signals & Slots

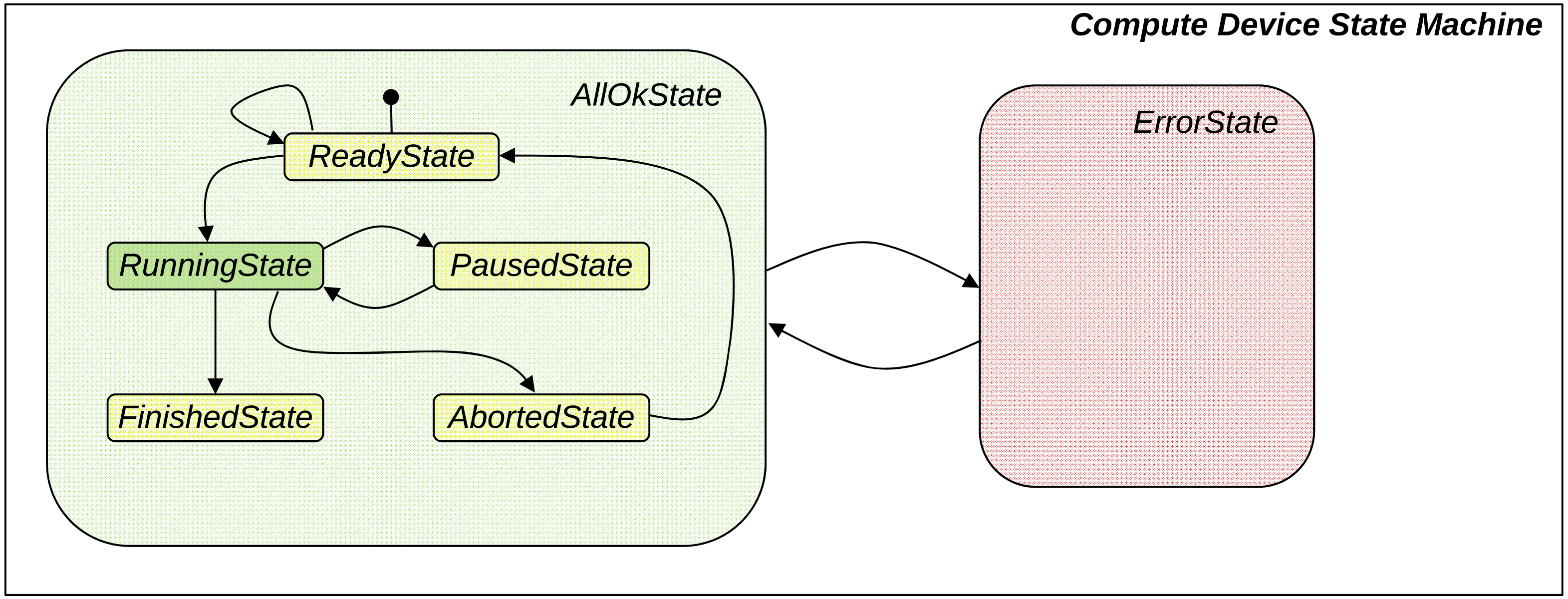
- Decoupling of the trigger of an action (signal) from the code that handles it (one or more slots)
- Simple expression of 1 x 1, 1 x N, N x 1 and N x N relationships
- Strictly event-driven system can be implemented (no polling needed)
- Developers are forced to implement to interfaces (signals and slots) in their components. This inherently structures and conventionalizes the whole communication layer
- Components are highly reusable and allow for composition/nesting

Flow-Control



- The communication is asynchronous and event-driven
  - Any slot may be called at any time without having influence on this
  - Different slots may be even called concurrently
- Sometimes we need some sequencing or synchronous behavior
  - E.g. The motor should move first to target position before I want to reconfigure the velocity
  - A request response pattern is needed and an error should be triggered if no one answers

Flow control – Using finite state machines



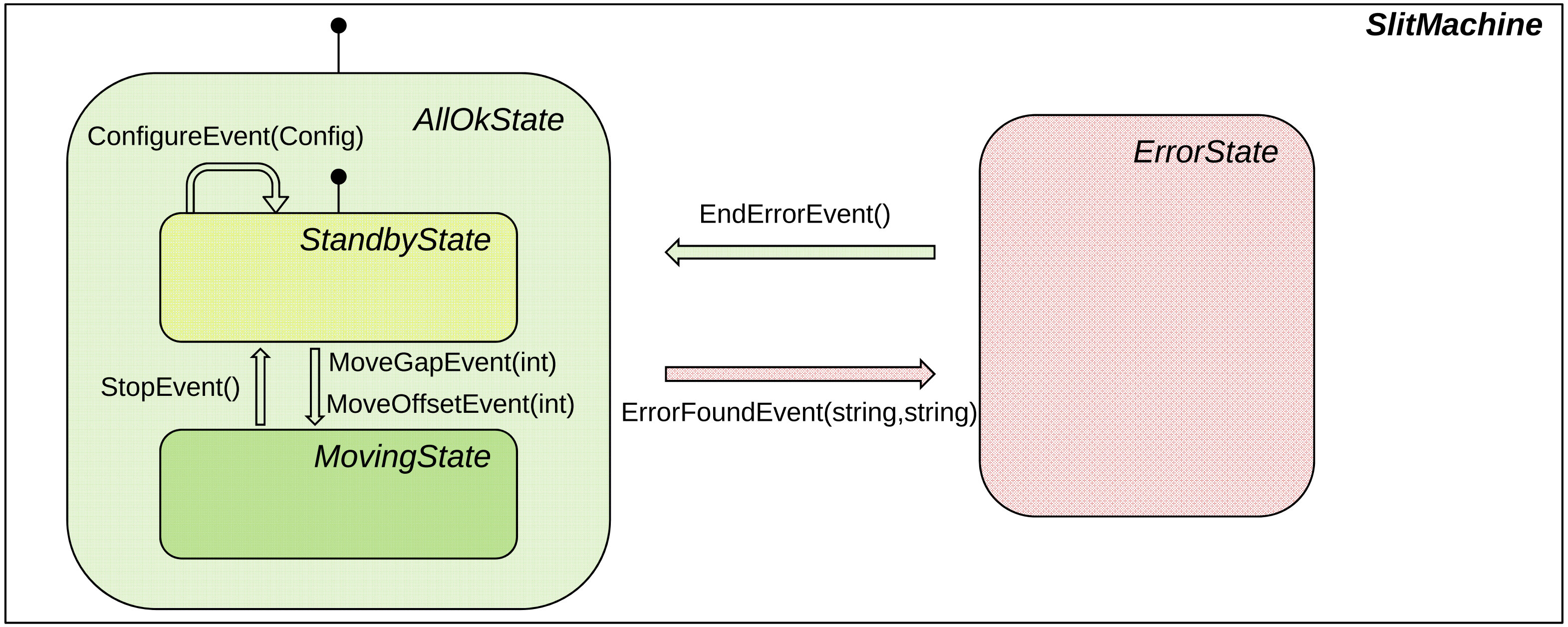
AllOkState

Source State	Event	Target State	Action	Guard
ReadyState	ReconfigureEvent	None	ReconfigureAction	IsValidReconfiguration
ReadyState	ComputeEvent	RunningState	ComputeAction	None
RunningState	PauseEvent	PausedState	PauseAction	None
RunningState	FinishedEvent	FinishedState	FinishAction	None

StateMachine

Source State	Event	Target State	Action	Guard
AllOkState	ErrorFoundEvent	ErrorState	ErrorFoundAction	none
ErrorState	EndErrorEvent	AllOkState	EndErrorAction	none

Flow control – Using finite state machines



AllOkState =

Source State	Event	Target State	Action	Guard
StandbyState	MoveGapEvent	MovingState	MoveGapAction	none
StandbyState	MoveOffsetEvent	MovingState	MoveOffsetAction	none
StandbyState	ConfigureEvent	none	ConfigureAction	none
MovingState	StopEvent	StandbyState	StopAction	NoMotorMovesGuard

SlitMachine =

Source State	Event	Target State	Action	Guard
AllOkState	ErrorFoundEvent	ErrorState	ErrorFoundAction	none
ErrorState	EndErrorEvent	AllOkState	EndErrorAction	none

European XFEL

Technical realization in C++


```cpp
// States
virtual void standbyStateOnEntry();
virtual void movingStateOnEntry();
FSM_STATE(ErrorState)
FSM_STATE_E(StandbyState, standbyStateOnEntry)
FSM_STATE_E(MovingState, movingStateOnEntry)
```

```cpp
// Events
FSM_EVENT2(ErrorFoundEvent, errorFoundEvent, string, string)
FSM_EVENT0(EndErrorEvent, endErrorEvent)
FSM_EVENT1(MoveGapEvent, slotMoveGapEvent, int)
FSM_EVENT1(MoveOffsetEvent, slotMoveOffsetEvent, int)
FSM_EVENT0(StopEvent, slotStopEvent)
FSM_EVENT1(ConfigureEvent, slotConfigureEvent, exfel::util::Config)
```

```cpp
// AllOkState Machine
struct AllOkStateTransitionTable : mpl::vector<
//      SrcState      Event           TgtState      Action           Guard
Row< StandbyState, MoveGapEvent    , MovingState , MoveGapAction   , none >,
Row< StandbyState, MoveOffsetEvent, MovingState , MoveOffsetAction, none >,
Row< MovingState , StopEvent       , StandbyState, StopAction      , noMotorMovesGuard > >(){};
//                 MachineName, InitialState, Context
FSM_STATE_MACHINE(AllOkState, StandbyState, SlitDevice)
```

```cpp
// SlitDevice Machine
struct SlitDeviceMachineTransitionTable : mpl::vector<
//     SrcState        Event           TgtState     Action          Guard
Row< AllOkState, ErrorFoundEvent, ErrorState, ErrorFoundAction, none >,
Row< ErrorState, EndErrorEvent  , AllOkState, EndErrorAction  , none > >(){};
FSM_TOP_MACHINE(SlitDeviceMachine, AllOkState, SlitDevice)
FSM_STARTUP(SlitDeviceMachine, startStateMachine)
```

```cpp
// Transition Actions
virtual void errorFoundAction(const string&, const string&);
virtual void endErrorAction();
virtual void stopAction();
virtual void moveGapAction(const int&);
virtual void moveOffsetAction(const int&);
virtual void configureAction(const exfel::util::Config&);
FSM_ACTION2(ErrorFoundAction, errorFoundAction)
FSM_ACTION0(EndErrorAction, endErrorAction)
FSM_ACTION1(MoveGapAction, moveGapAction)
FSM_ACTION1(MoveOffsetAction, moveOffsetAction)
FSM_ACTION0(StopAction, stopAction)
FSM_ACTION1(ConfigureAction, configureAction)
```

```cpp
// Guards
bool noMotorMovesGuard();
FSM_GUARD0(NoMotorMovesGuard, noMotorMovesGuard)
```

- Reflects the full implementation of the state machine
- Events that should be trigger-able from outside are just made Slots

Clip board – copy and paste

Headline

- first level
    - second level
        - → third level

Headline

Texttext texttext texttext texttext texttext texttext

Keyword

1. Keyword
2. Keyword

- keyword
- keyword

Result Headline

- result text
- result text

Result headline

Result text, result text, result text

Result headline

- result text
- result text
- result text