Odysseus
the event processing system

# An Extensible Research Platform for Streaming Applications

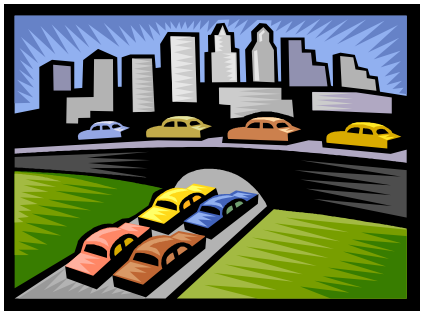
Marco Grawunder, University of Oldenburg

Databases and Informationssystems
Department of Computing Science
University of Oldenburg
http://www. uni-ol.de/is/

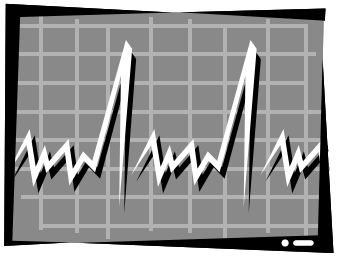
# Outline

- Motivation for DSMS
- Odysseus Input
  - Adapter Framework
- Odysseus Processing
  - Internal Model
  - Processing of Windows
- Odysseus Provisioning
  - Adapter Framework
- Odysseus and SPARQL/RDF
- Odysseus Architecture

# New Applications



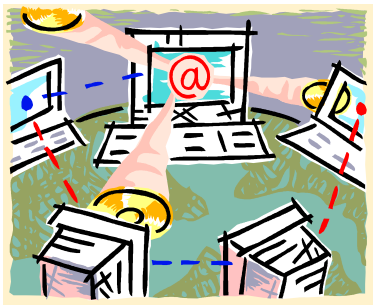
Traffic Management



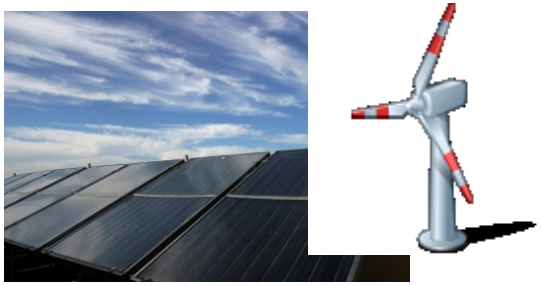
Medical Monitoring

Logistics

Electronic Trading

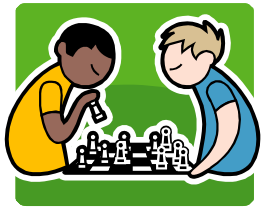


Network management



Energy management

Industry 4.0

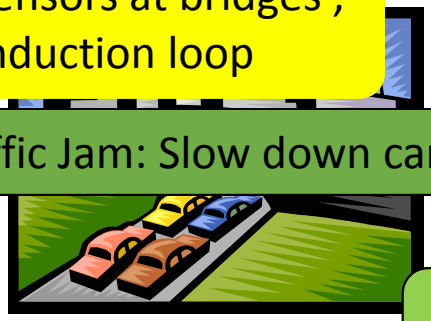


Games

# New Applications

sensors at bridges , induction loop

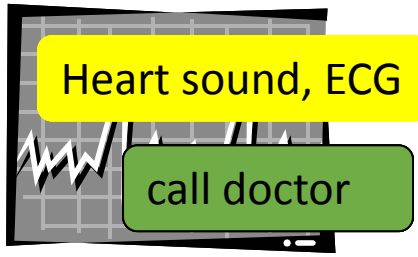
Traffic Jam: Slow down cars



Traffic Management

Heart sound, ECG

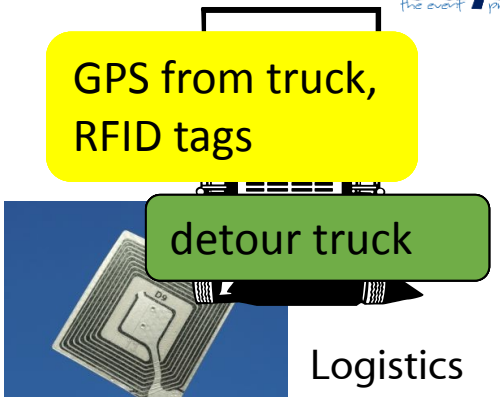
call doctor



Medical Monitoring

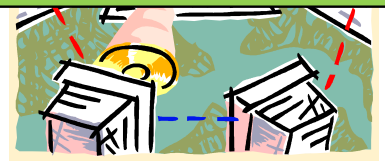
GPS from truck, RFID tags

detour truck



Logistics

Active sources/sensors deliver continuously data/events (→ data-/event stream)



Network management

Energy consumption and production

Stock exchange price, sells, buys

Sell or buy

Electronic Trading

react in front of heavy failures

Games

Necessary: Fast reaction in critical situations

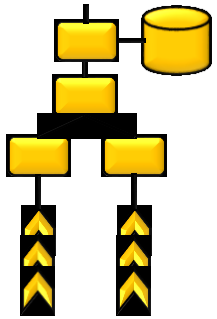
Industry 4.0

# Way to process event streams

- Hardware based
  - efficient
  - not very flexible
  - high modification effort
- Write you own program
  - efficient
  - quite flexible
  - Maintenance may be a problem, adaptability for new problems?
  - error-prone
  - bad scalability

```
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
try {
  switch ((jj_ntk==-1)?jj_ntk()
  case 66:
    jj_consume_token(66);
    CompositeSQLStatement();
    jj_consume_token(67);
    break;
  case K_SELECT:
    SelectClause();
    FromClause();
```

- Use a System/Framework: Data stream management systems (DSMS)/CEP
  - efficient
  - flexible
  - easy maintenance and adaptability
  - less error-prone
  - good scalability



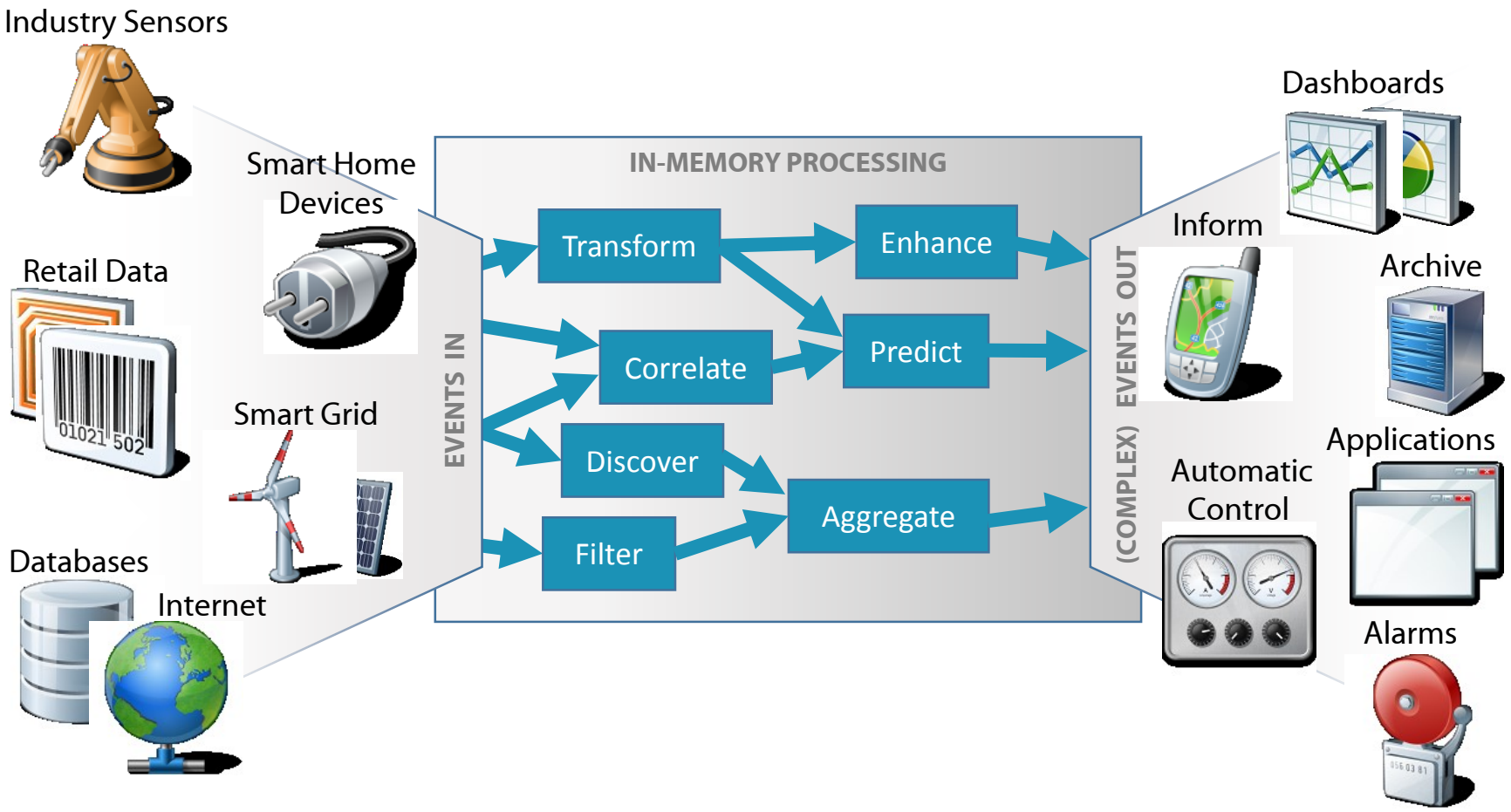
# Odysseus

the event processing system

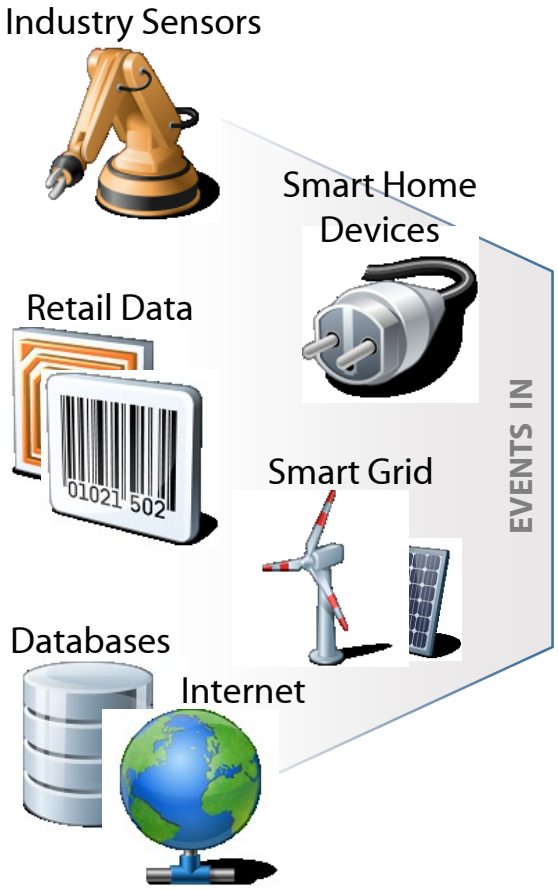
## Concepts

- Start of project: May 2007
- Goal: Development of a software platform to evaluate data stream processing method
  → Extensibility and easy maintenance always a main concern
- Many redesigns → gained high experience
- Extensible plug-in architecture (Java with OSGi)
- Today:
  - Mostly stable code base (the core)
  - About 3 Mio. „Lines of Code“, about 400 plug-ins
  - Five completed dissertations, about 5 currently active
  - ~35 bachelor thesis, ~25 master thesis
  - Many student project groups
- Current developers
  - about 5-10 research assistants (not all providing core stuff)
  - Multiple students
  - Some external developer
- Apache 2 Licence (for most parts)

# Our basic Scenario



# Our basic Scenario

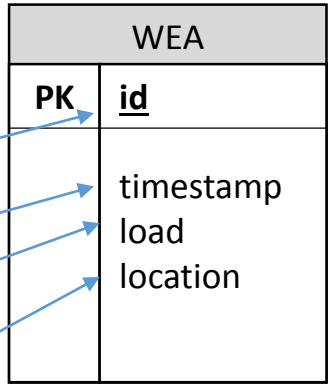



Sources | Integration | Processing | Provisioning | Sinks

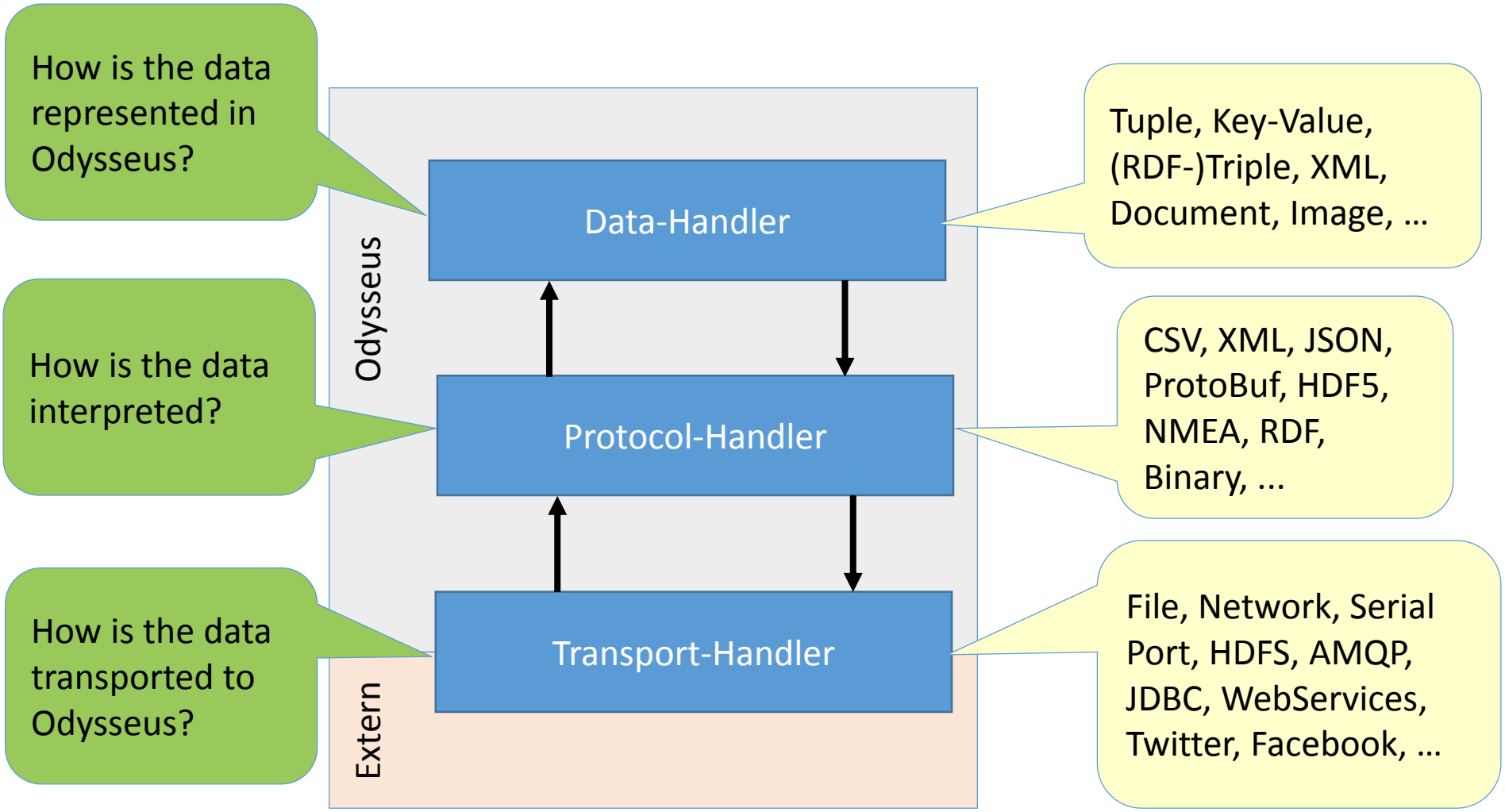
# Example Source definition: wind park

```
#PARSER PQL
#RUNQUERY
wea := RECEIVE({

                ['id', 'Integer'],
                ['timestamp', 'StartTimestamp'],
                ['load', 'Double'],
                ['location', 'SpatialPoint']
            ],


        }
    )
```

Time stamp!

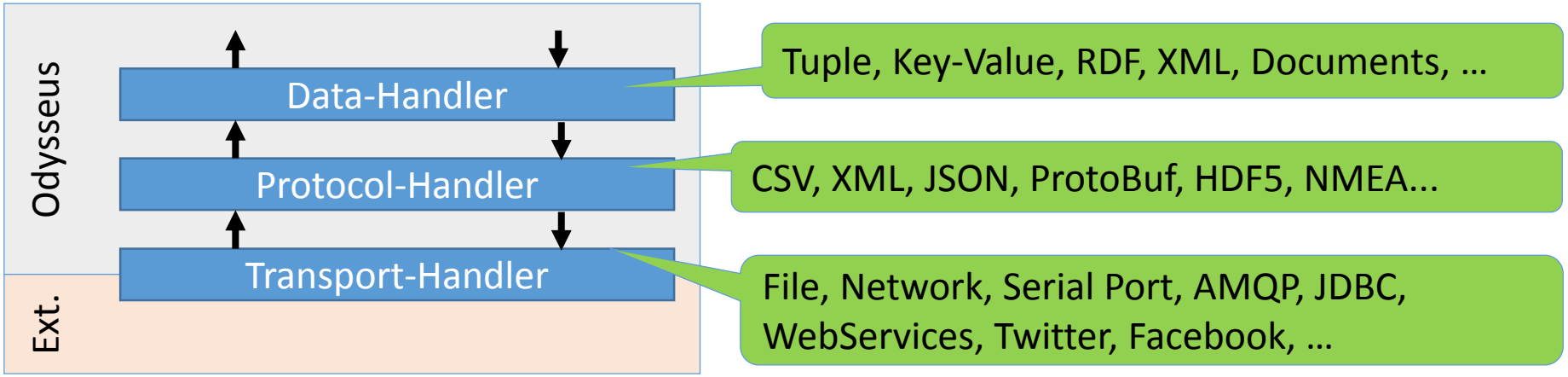


# Adapter-Framework



# Define sources

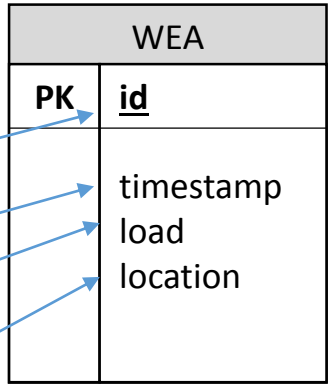
- Simple to add sources to the system
  → Adapter framework
- Essential: Easy extensibility and combinability
- Many are bidirectional (for sources and sinks)
- Currently available:
  - more than 800 Adapter by combing protocol and transport
  - Multiple internal representations (data handler)
- Selectable/Combinable by user in query language
- Extensible and configurable



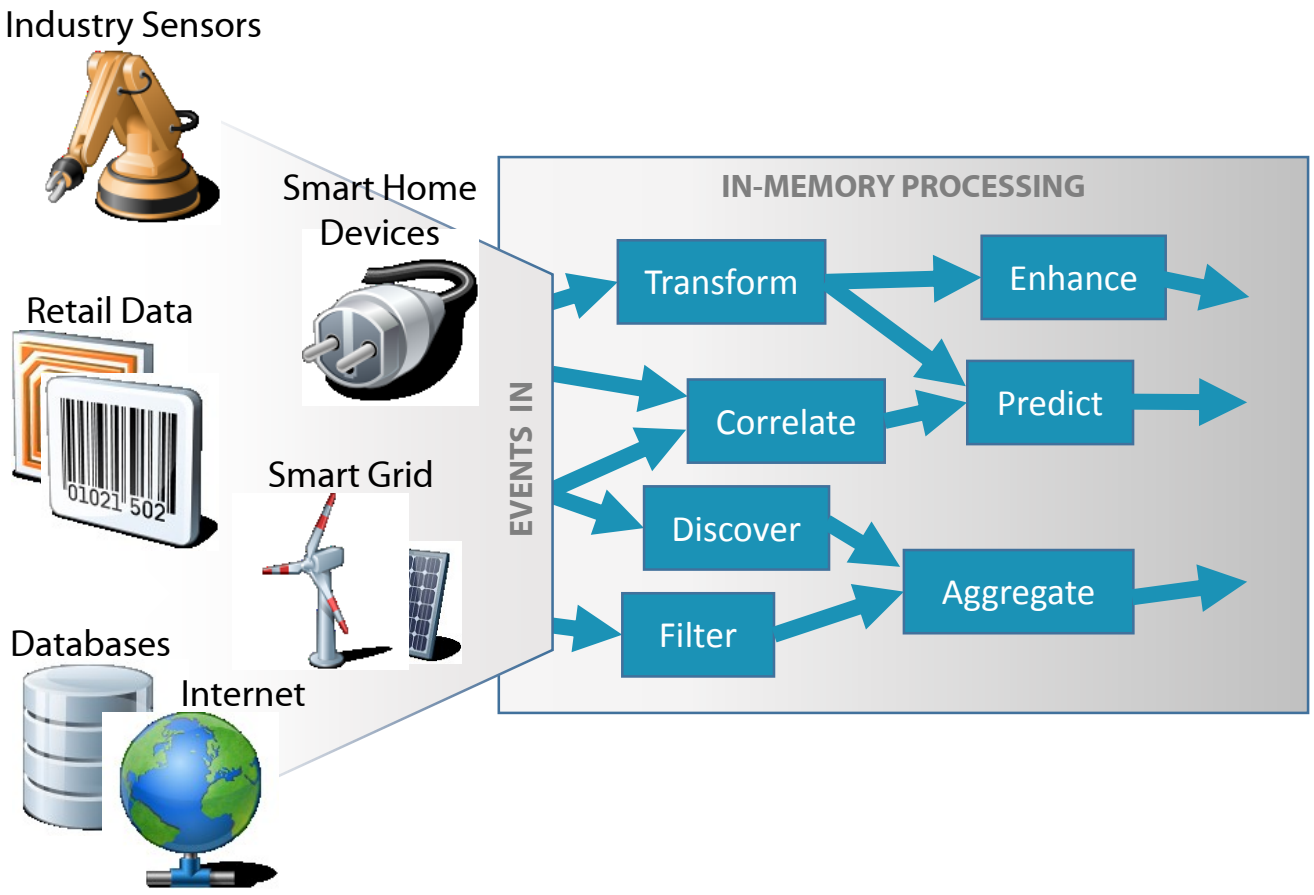
# Example Source definition: wind park

```
#PARSER PQL
#RUNQUERY
wea ::= RECEIVE({
            source = 'wea',
            transport = 'tcpclient',
            datahandler = 'tuple',
            protocol = 'simplecsv',
            schema = [
              ['id', 'Integer'],
              ['timestamp', 'StartTimestamp'],
              ['load', 'Double'],
              ['location', 'SpatialPoint']
            ],
            options = [
              ['host', '123.0.1.2'],
              ['port', '1230']
            ]
          }
        )
```

Time stamp!



# Our basic Scenario

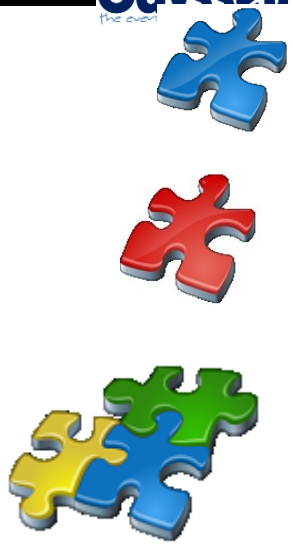


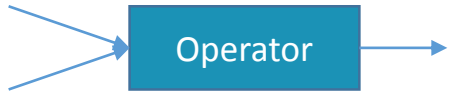

Sources | Integration | **Processing** | Provisioning | Sinks

# Odysseus as processing platform

- Operator based
  - Enrichment, analysis and correlation of data
  - read input streams and produce output stream
  - independent!

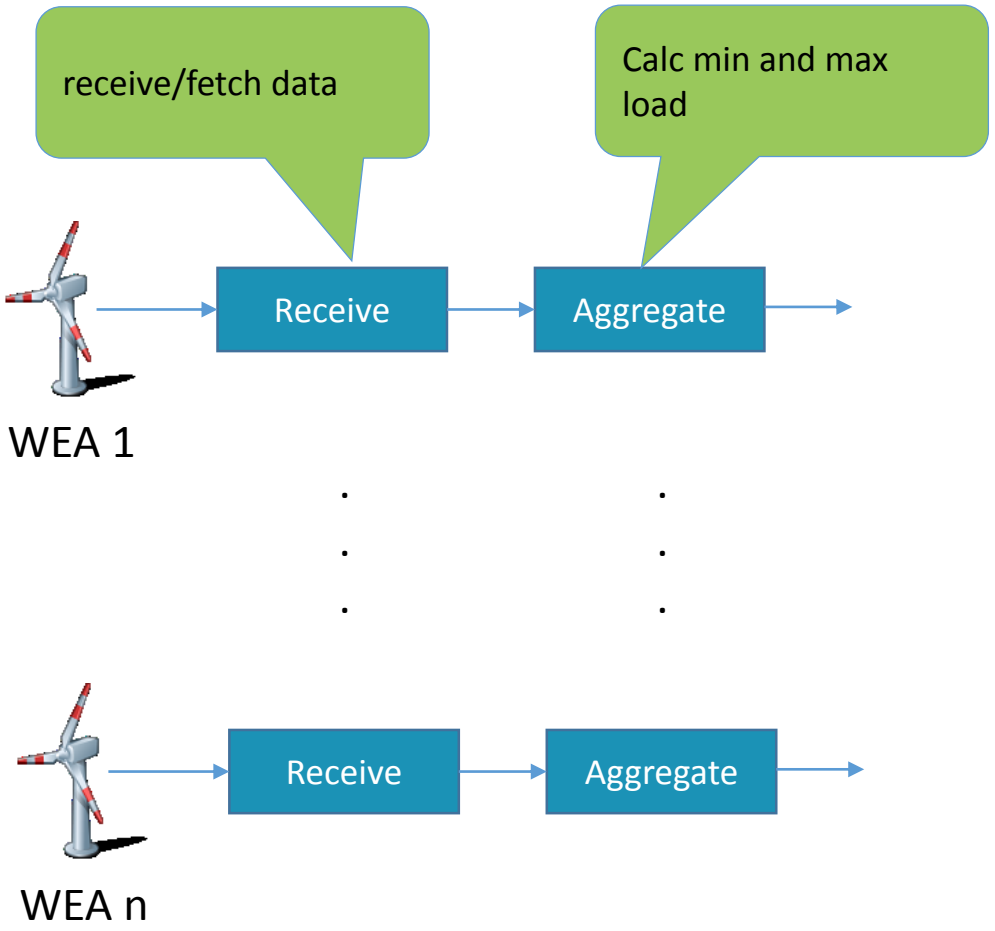
➔easy combination to complex queries





# Example: Lastgang (min und max load)

| WEA | |
|---|---|
| **PK** | **id** |
| | timestamp<br>load<br>location |



# How to express event processing?

- Language?
- Level of abstraction?
- Typical problem:
  - for special processing new operators are needed
  - Problem: How to handle extensibility without touching the query parser?
  - More worse: In plug-in based systems, the parser must be independent of other plug-ins
- Possible approach:
  - Declarative query language like SQL, CQL, SPARQL
  - Allow only functions for new operations
  - Plug-ins register their functions in the main systems
  - Use this function in SELECT and WHERE part of a SQL like query

# How to express event processing?

- Odysseus has a CQL based language … but we do not use it really
- Problems:
  - Some operations are not intuitive expressible as simple functions
  - Often it is easier to look at the data/event flow: Define what to do with the incoming events
  - Sometime additional options are needed
- Odysseus special approach (quite similar to PIG latin)
  - Use an operator based language with a general syntax
  - Operators define the flow of events
  - Plug-ins can define new operators
  - → PQL

# Query Language PQL

- PQL-Approach:
  - data flow based query language
  - Format:
    **OP({<parameter set>},source$_1$, source$_2$, ..., source$_N$)**
  - Automatic generation from operators (by Java annotations)

- Example: Lastgang

```
#PARSER PQL
#ADDQUERY
out = AGGREGATE({
          aggregations=[
              ['MAX', ,load', 'max_load', 'double'],
              ['MIN', ,load', 'min_load', 'double']
          ]
        },
        wea
      )
```

# Query Language PQL

- PQL-Approach:
  - data flow based query language
  - Format:
    `OP({<parameter set>},source1, source2, ..., sourceN)`
  - Automatic generation from operators (by Java annotations)

- For a new operator we need:
  - A logical representation (an algebra operator)
  - A physical representation (the executing operator)
  - A transformation rule: Under which circumstances should we which operator used
    - E.g. transformation of a filter/selection in tuple based scenarios is different to key-value based
  - Current Work: Replace all above with a single definition

# Annotation based logical operators

```
@LogicalOperator(name = "AGGREGATE", minInputPorts = 1, maxInputPorts = 1, …)
public class AggregateAO extends UnaryLogicalOp implements IStatefulAO {
…

@Parameter(name = AGGREGATIONS, type = AggregateItemParameter.class, isList = true)
public void setAggregationItems(List<AggregateItem> aggregations) {
        …
}
```

```
#PARSER PQL
#ADDQUERY
out = AGGREGATE({
        aggregations=[
            ['MAX', ,load', 'max_load', 'double'],
            ['MIN', ,load', 'min_load', 'double']
        ]
    },
    wea
)
```

# Annotation based logical operators

```
@LogicalOperator(name = "AGGREGATE", minInputPorts = 1, maxInputPorts = 1, …)
public class AggregateAO extends UnaryLogicalOp implements IStatefulAO {
…

@Parameter(name = AGGREGATIONS, type = AggregateItemParameter.class, isList = true)
public void setAggregationItems(List<AggregateItem> aggregations) {
        …
}
```

```
#PARSER PQL
#ADDQUERY
out = AGGREGATE({
        aggregations=[
            ['MAX', ,load', 'max_load', 'double'],
            ['MIN', ,load', 'min_load', 'double']
        ]
    },
    wea
)
```

# Problems of a potentially infinite event stream

# Problems of a potentially infinite event stream

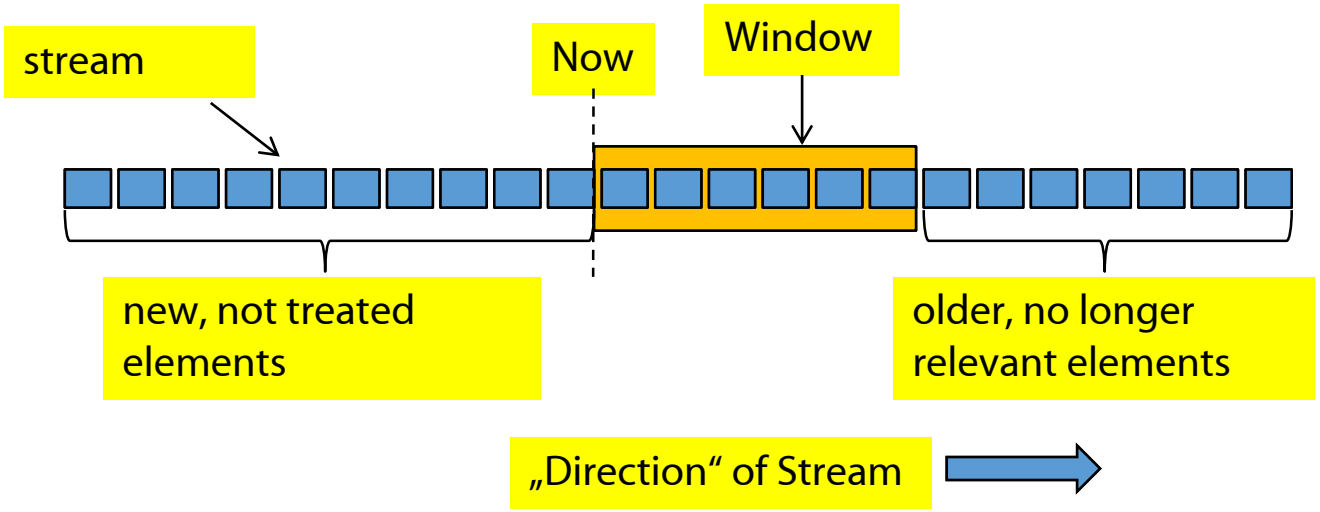
- In the example
  - Min/Max goes back until query start
  - Maybe, this is not the information currently needed?
- More worse: Could lead to OutOfMemoryException


© Michael Wolf, http://www.photomichaelwolf.com/tokyo_subway_dreams/

# Problems of an infinite data stream

- Typically, only the current history relevant
  - Example: the last 15 minutes

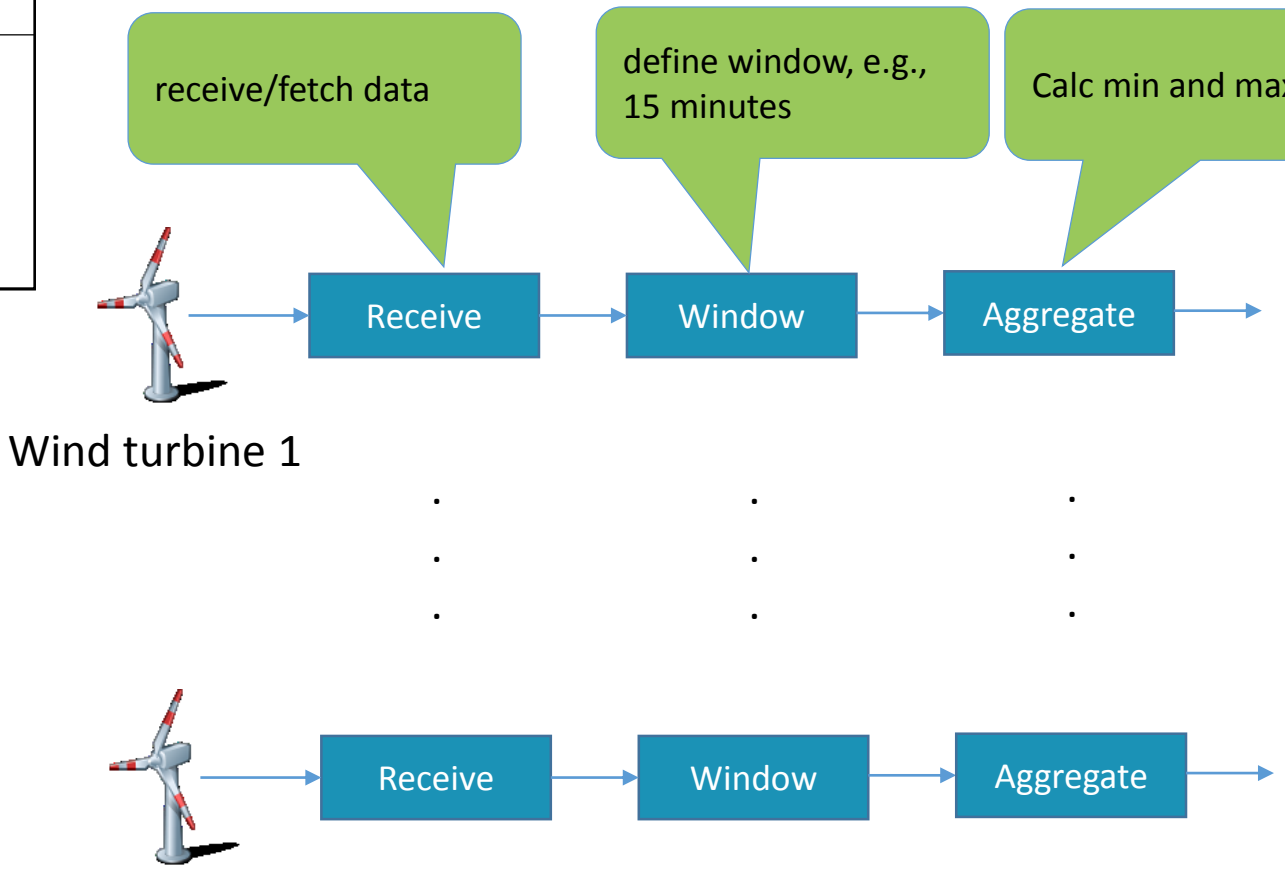
→ Window: Only look over an stream excerpt




- Windows (in Odysseus) can be based on
  - the time, the number of elements, on predicates

# Example: Lastgang

| WEA | |
|---|---|
| **PK** | **id** |
| | timestamp<br>load<br>location |



# Window processing in Odysseus

- Necessary: Way to describe which elements are in the same window, i.e. must be processed together

Window

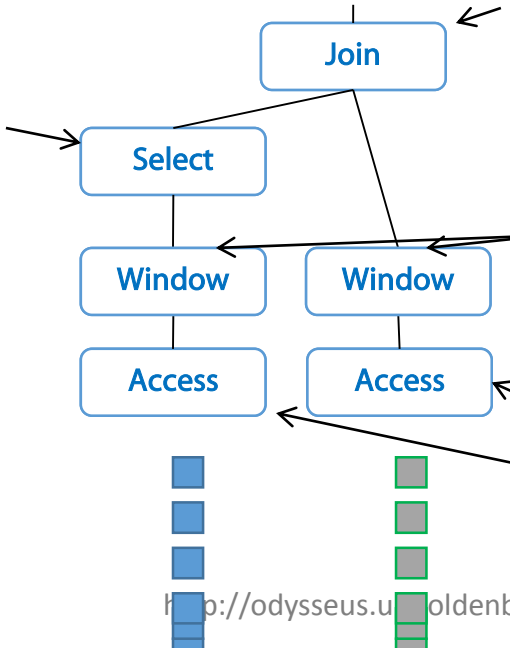
- Local windows:
  - Each operator (aggregation, join, etc.) defines its own window (storage)
  - Each operator must implement window processing for time based, element based and predicate based windows
  - Could lead to problems with semantics and determinism
- Global window:
  - each stream element has additional meta data
  - Operator use meta data to decide which elements build a window
  - Two typical approaches:
    - Pos/Neg: annotate each element with timestamp and marker if its a start (+) or end timestamp (-)
    - Interval approach: annotate each element with start and end time stamp

# Odysseus: Interval approach

- Each element gets a right open valid time interval $[t_s,t_e)$
- Elements reaching system: Set start time stamp $t_s$
  - with transaction time: Current system time, or
  - with application time: Retrieve value from input (e.g. attribute with time stamp of sensor)
- Window operators set end time stamp
- Some Operators need to look at the timestamps, but need not to care about window type

Join: Merging of elements in the same window, i.e. valid time interval overlap

Selection: without looking at valid time interval

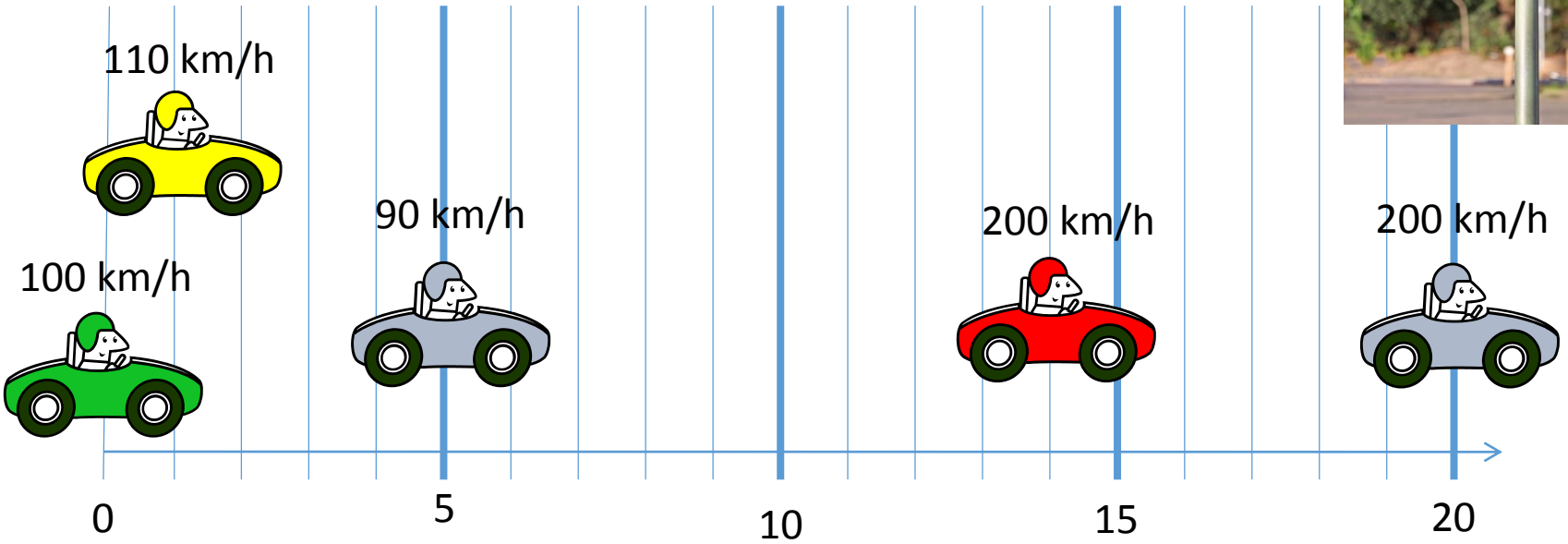



Window operator
Set end time stamp: $\{e, [t_s,t_e)\}$
Depends on window type and size

Access temporal ordered input,
Set start time stamp: $\{e, [t_s,\infty)\}$

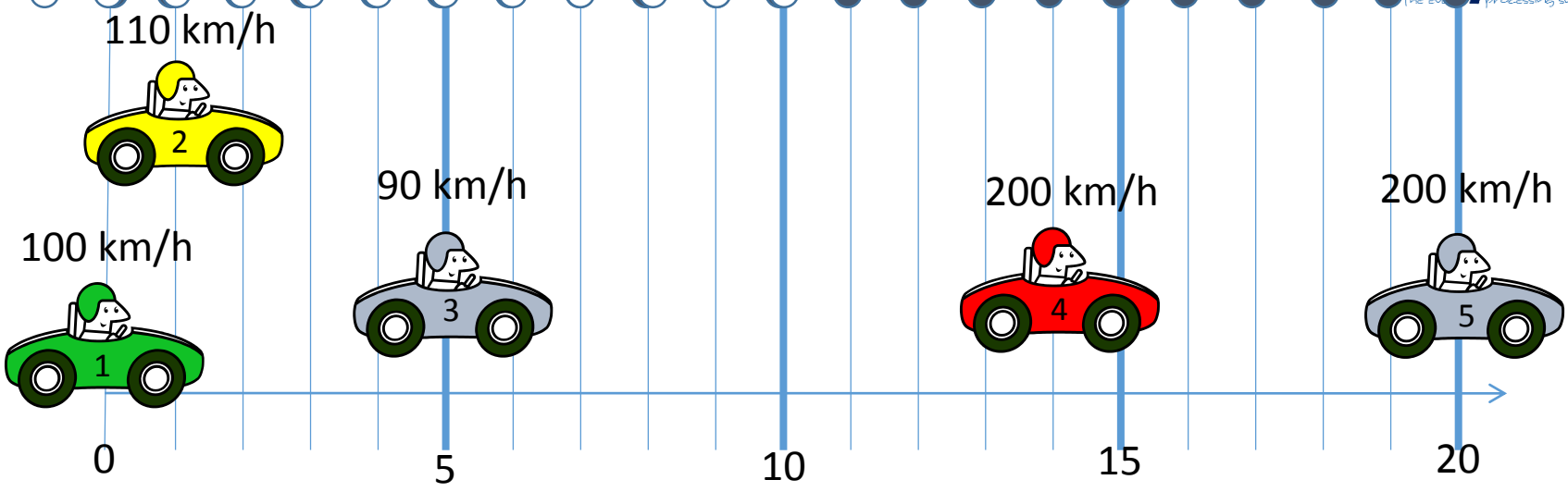
# How to process aggregations?

- Example: Average speed over the last 10 minutes




Quelle: http://kanzlei-blog.de/geschwindigkeitsmessung-kurz-vor-dem-ortsausgangsschild/

# With 10 minutes Sliding Window




| | | |
|---|---|---|
| 16:00 | 1 | 100/1 = 100 |
| 16:01 | 1,2 | (100+110)/2 = 105 |
| 16:02 | 1,2 | 105 |
| 16:03 | 1,2 | 105 |
| 16:04 | 1,2 | 105 |
| 16:05 | 1,2,3 | (100+110+90)/3 = 100 |
| 16:06 | 1,2,3 | 100 |
| 16:07 | 1,2,3 | 100 |
| 16:08 | 1,2,3 | 100 |
| 16:09 | 1,2,3 | 100 |
| 16:10 | 2,3 | (110,90) = 100 |

| | | |
|---|---|---|
| 16:11 | 3 | 90/1 = 90 |
| 16:12 | 3 | 90 |
| 16:13 | 3 | 90 |
| 16:14 | 3,4 | (90+200)/2 = 145 |
| 16:15 | 4 | 145 |
| 16:16 | 4 | 145 |
| 16:17 | 4 | 145 |
| 16:18 | 4 | 145 |
| 16:19 | 4 | 145 |
| 16:20 | 4,5 | (200+200) / 2 = 200 |
| 16:21 | 4,5 | 200 |

# How to calculate aggregates?

- The easy way:
  - Keep all elements
  - When new element gets into the operator, add it and remove all now invalid element
    (i.e. when the start time stamp < newElement – window size)
  - For each new input create output
  - We do this for median calculation
- Problems:
  - Each element must be stored!
    - What if the window is defined over 30 days?
  - Need to handle "in-between" values: e.g., at 16:11 only one car
- Odysseus Approach: Use partial aggregates
  - Keep only information that is necessary:
    For average only count and sum is needed
  - Use init, merge and eval functions on partial aggregates
- Use time interval annotations to represent different states

# PQL Query in Odysseus

```
#PARSER PQL
#ADDQUERY
in = CSVFILESOURCE({
          delimiter = '\t',
          schema = [
            ['ts', 'STARTTIMESTAMP'],
            ['carID', 'Integer'],
            ['kmh','Integer']
          ],
          source = 'cars',
          filename = '${WORKSPACEPROJECT}/cars.csv',
          options = [
                  ['baseTimeUnit','MINUTES']
          ]
        }
      )

win = TIMEWINDOW({SIZE = [10, 'MINUTES']}, in)

agg = AGGREGATE({
          aggregations = [
            ['AVG', 'kmh', 'avg_kmh'],
            ['NEST','carID','cars']
          ]
        },
        win
      )
```

Define simple input source (here csv)

cars.csv

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 100 |
| 2 | 1 | 2 | 110 |
| 3 | 5 | 3 | 90 |
| 4 | 14 | 4 | 200 |
| 5 | 20 | 5 | 200 |

What is the time unit of the input?

A window over 10 minutes

Calc aggregation and a nesting

# Output in Interval Approach

| avg_kmh | cars | start | end |
|---|---|---|---|
| 200.0 | [5] | 24 | 30 |
| 200.0 | [4, 5] | 20 | 24 |
| 200.0 | [4] | 15 | 20 |
| 145.0 | [3, 4] | 14 | 15 |
| 90.0 | [3] | 11 | 14 |
| 100.0 | [2, 3] | 10 | 11 |
| 100.0 | [1, 2, 3] | 5 | 10 |
| 105.0 | [1, 2] | 1 | 5 |
| 100.0 | [1] | 0 | 1 |

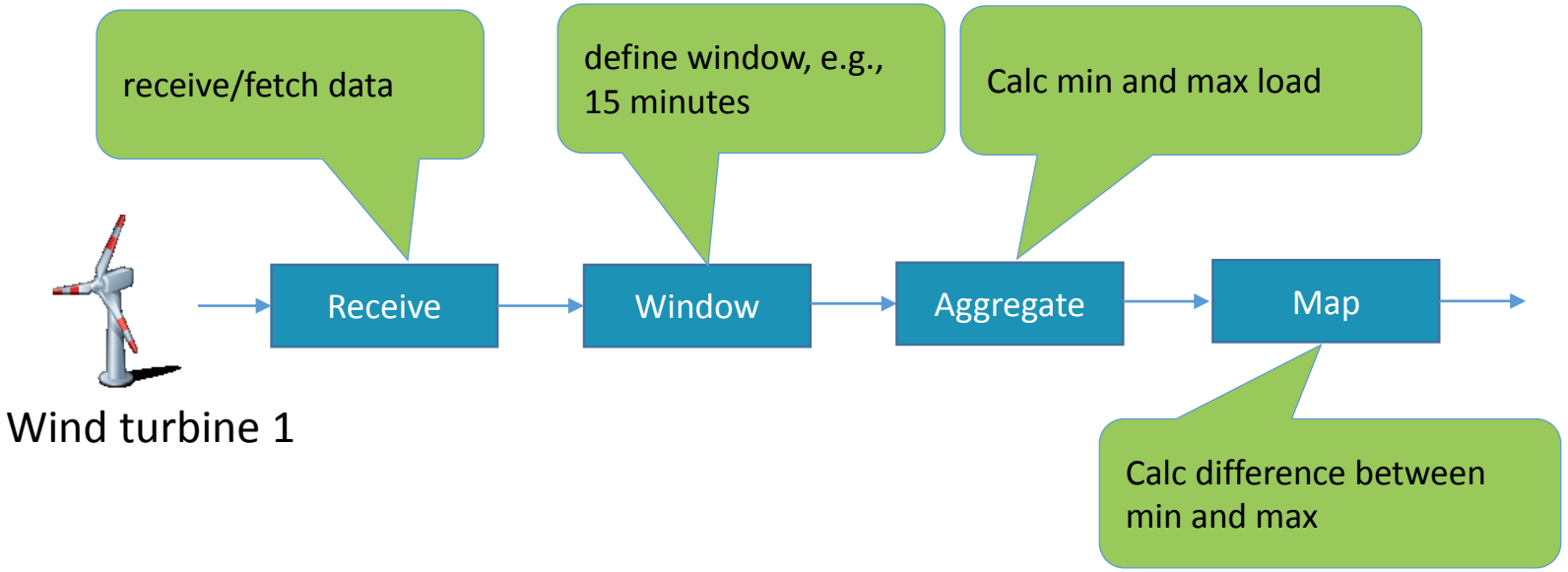
- e.g.
  - 100 km/h with car 1 is valid from 0 to 1
  - 105 km/h with cars 1 and 2 is valid from 1 to 5
  - 90 km/h with car 3 is valid from 11 to 14
  - ...
- Handling of correct intervals is very important for the right semantic!
- Compressed output by intervals (instead of timestamps for every beat)

# Data transformation

- Data transformation
  - Map-Operator:
    - mathematical expressions and functions over single attribute
    - Result typically a tuple (for most functions)

More than 250 mathematical functions available – easy extendible

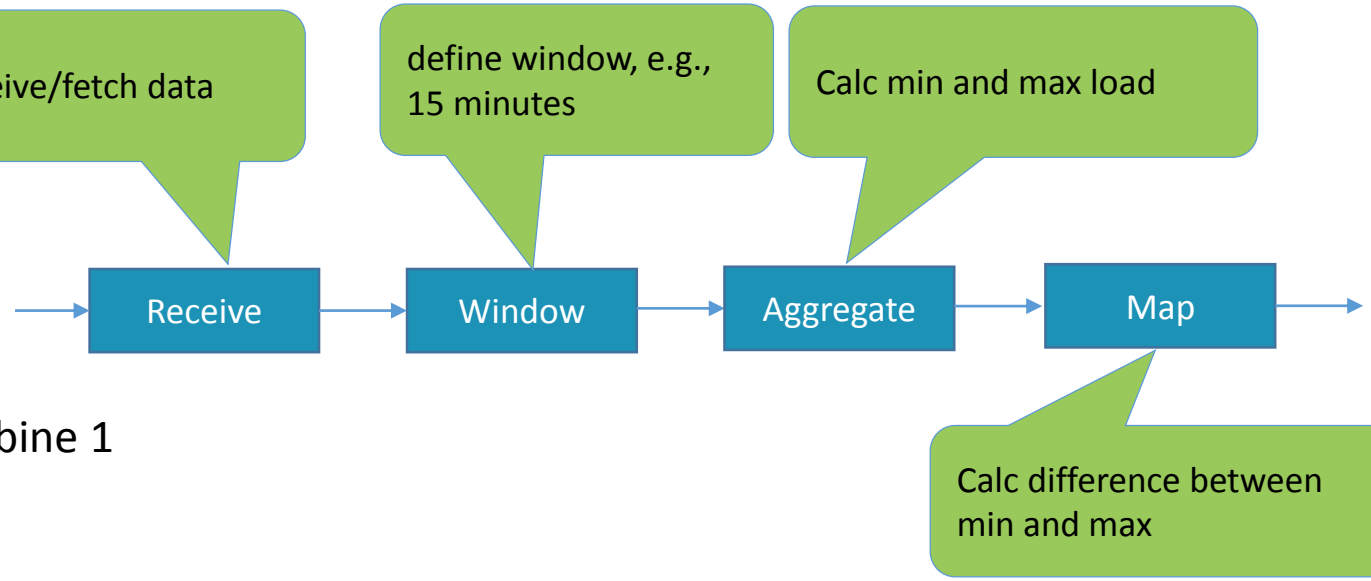
- Example: Spread of Lastgang



# Data transformation

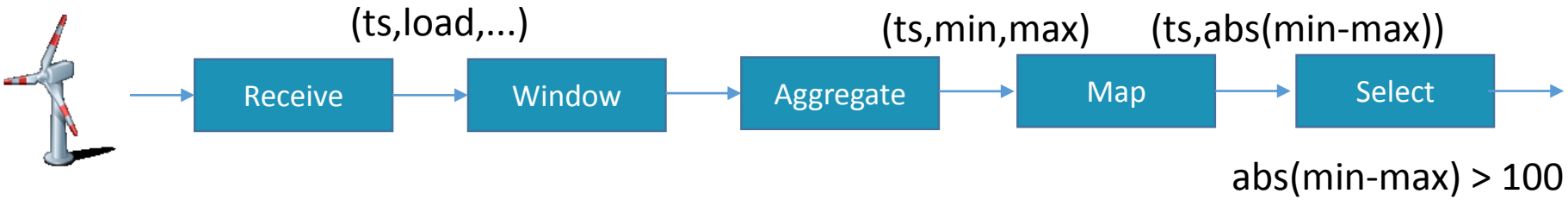
```
spreizung = MAP({
                expressions=[
                    'id',
                    'max_load',
                    'min_load',
                    ['abs(min_load-max_load)', 'lastspreizung']
                ]
            },
            lastgang
        )
```

Wind turbine 1



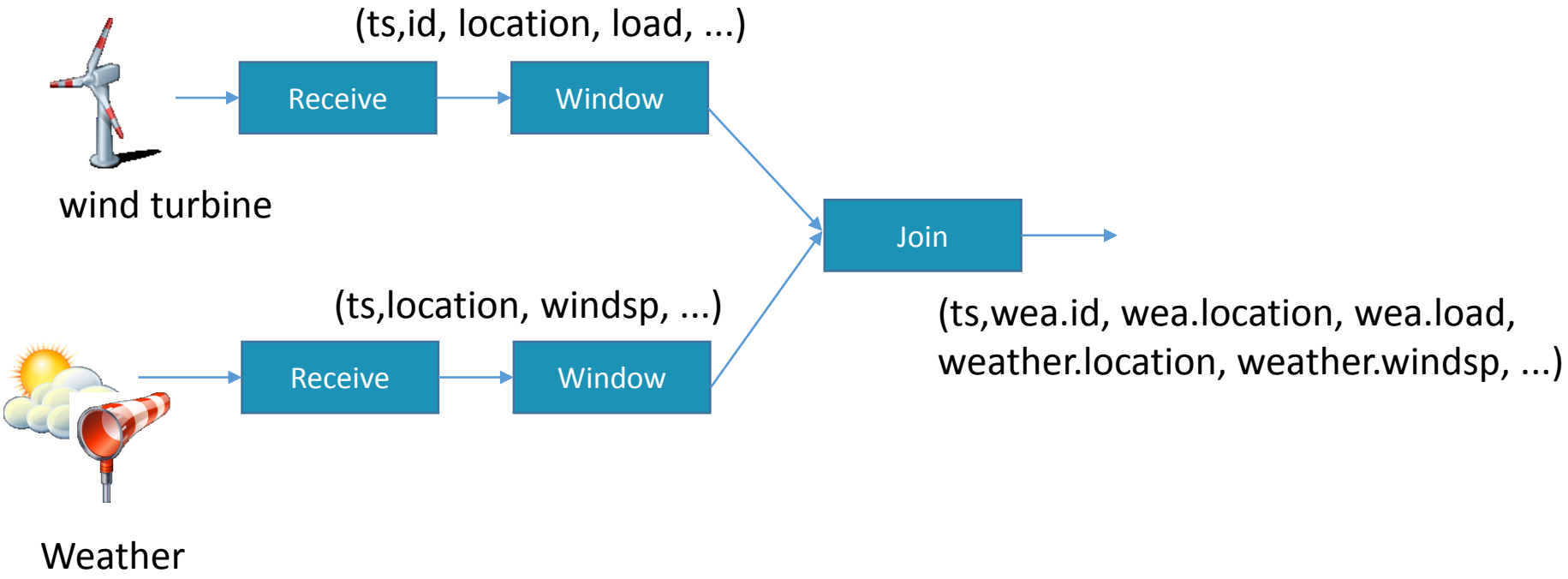
# Filter/Selection

- Filter (Selection)
  - Filter non interesting event
  - Send element to next operator only when spread > 100



# Join/Enrich

- Join weather data with data of a wind turbine based on geographical coordinates



# Scenario: PV-Condition-Monitoring

$U_{DC}$

$I_{DC}$

$P_{AC}$

Solarization

Receive

Receive

Receive

Receive

Join

$U_{DC} > U_{max}$ AND
$E_{min} <$ Solarization AND
Solariization $< E_{max}$

Select

Alarm:
Inverter overloaded

Select

Alarm: Site irregular condition

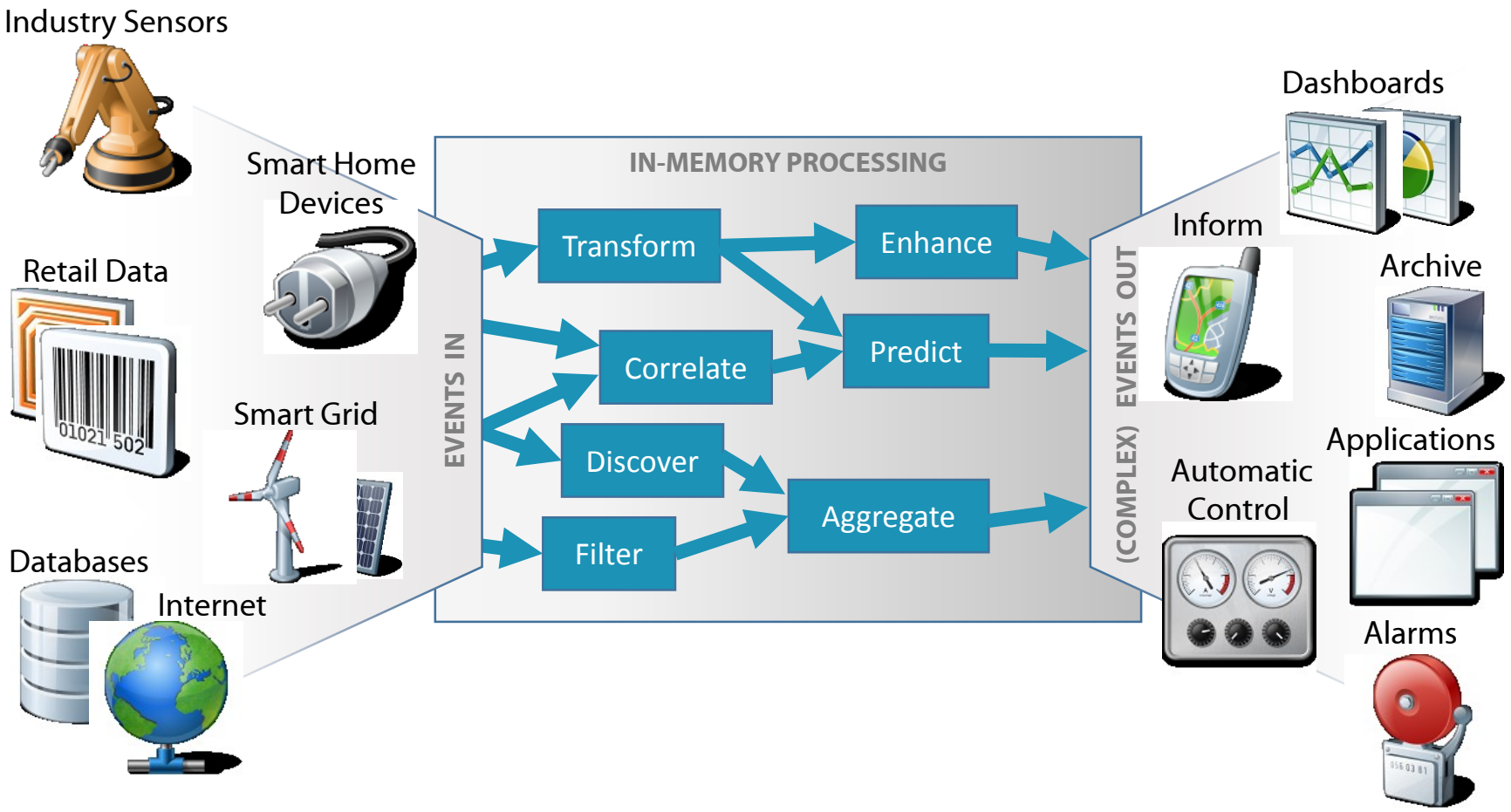
NOT($U_{DC} > U_{ein}$ AND
$I_{min} < I_{DC}$ AND
$I_{DC} < I_{max}$ AND
$E_{min} <$ Solarization AND
Solarization $< E_{max}$ AND
$P_{min} < P_{AC}$ AND
$P_{AC} < P_{max}$)

Fokus on **Know-How**, not on programming!

# Odysseus as processing platform

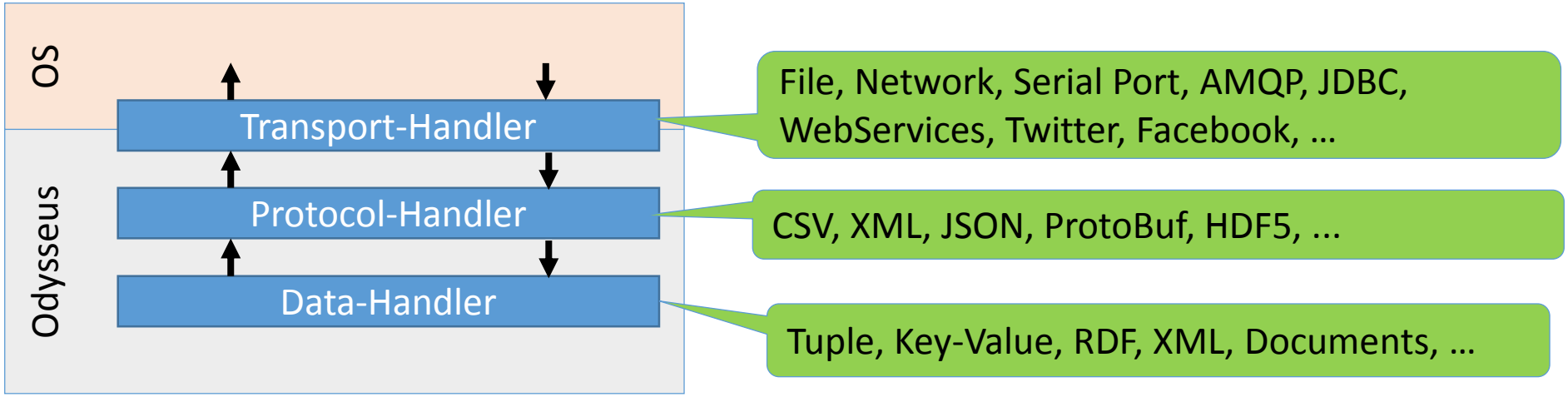
- Standard operators
  - Filter (Selection)
  - Transformation (Map)
  - Projection
  - Join
  - Aggregation (with MIN, MAX, AVG, MEDIAN, NEST, ...)
  - Set operations: Union, Intersection
- Further operators (excerpt)
  - Pattern matching (e.g. SASE+)
  - Classification, Regression, Clustering, …
  - Enrichment, …
  - Web service- and data base access,…
- Simple extension with new operators and other functions

# Our basic Scenario



# Provisioning with adapter framework

- Provider data for applications
- Alarm functions (SMS, E-Mail, ...)
- Adapter level analog to data integration



# Odysseus

the event processing system

## Odysseus and RDF/SPARQL

# Streaming SPARQL

- One of our first works with Odysseus (about 2007/8)
- In early years with special operators for RDF/Triples (e.g. BasicGraphSelection for Filter)
- Now: Just one additional operator TriplePatternMatching (for Basic Graph Pattern): reads Triples, writes Tuples
- Source definition with Odysseus access framework (using sesame for reading rdf input) → many other sources possible
- Slight modification on Filter syntax necessary (example later)
- OPTIONAL currently not working (Outer Joins are missing atm)
- Not all RDF functions working
- New: Window definition in SPARQL
  - Over sources in the FROM Part → Typical in other approaches
  - Over Basic Graph Pattern (TriplePatternMatching) → Needed because different content (e.g. load and rotation measurement) is sent over the same source

# SPARQL Example: Creating Stream Sources

```
#PARSER PQL
#ADDQUERY
recShop := RDFSOURCE({
          transport = 'file',
          source = 'RecShop',
          wrapper = 'GenericPull',
          rdf.format = 'RDFXML',
          options = [
            ['filename','${WORKSPACEPROJECT}/simple.rdf']
          ]
        }
      )
```

A file based source

Name of the source (to store in data dictionary)

Retrieve values from file by pull

What rdf serialization type (from sesame rio)

Options depending on transport

# SPARQL Example: Creating Stream Sources

```
#ADDQUERY
webtest := RDFSOURCE({
          transport = 'HTTP',
          source = 'webtest',
          wrapper = 'GenericPull',
          rdf.format = 'RDFXML',
          options = [
            ['uri','http://bioimages.vanderbilt.edu/baskauf/11409.rdf'],
            ['method','get']
          ]
        }
      )
```

A http based source

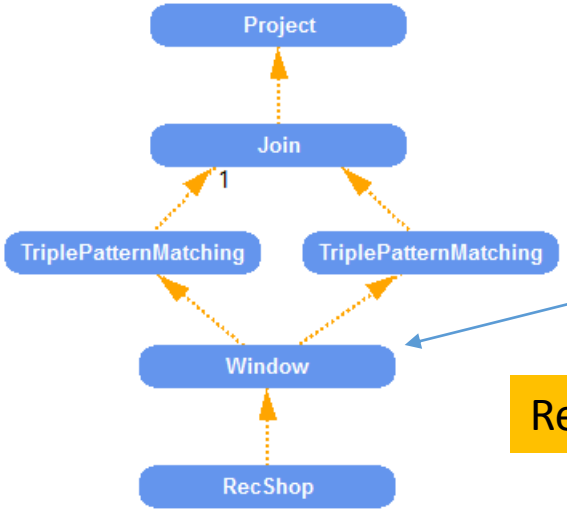
Options depending on transport

# SPARQL Query

```
#PARSER SPARQL
#ADDQUERY
PREFIX cd: <http://www.recshop.fake/cd#>
SELECT ?x ?y ?price
FROM STREAM <System.recShop> WINDOW RANGE 10 MS ADVANCE 10 MS
WHERE {?x cd:price ?price.
       ?x cd:artist ?y }
```

New Keyword STREAM: Use Odysseus Sources

Create a WINDOW over Stream




Window

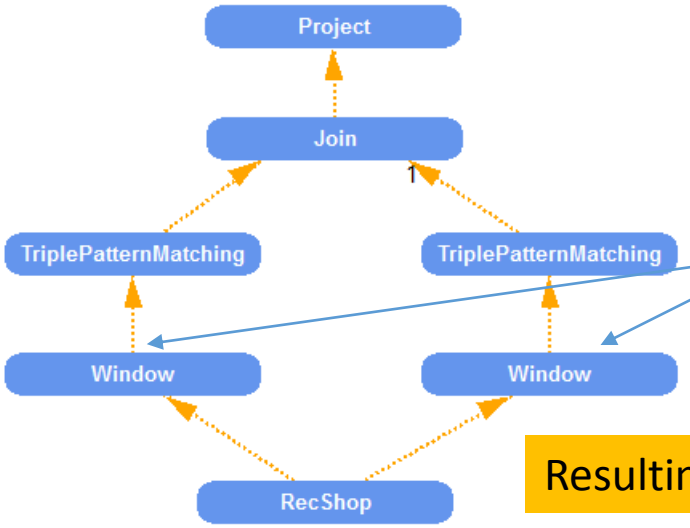
Resulting Query Execution Plan

# Window on BGP

```
#PARSER SPARQL
#ADDQUERY
PREFIX cd: <http://www.recshop.fake/cd#>
SELECT ?x ?y ?price
FROM STREAM <System.recShop>
WHERE {{?x cd:price ?price WINDOW RANGE 10 MS ADVANCE 10 MS} .
       {?x cd:artist ?y WINDOW RANGE 20 MS ADVANCE 10 MS} }
```

Additional Brackets needed

Create WINDOWs over Basis Graph Pattern




Two Windows

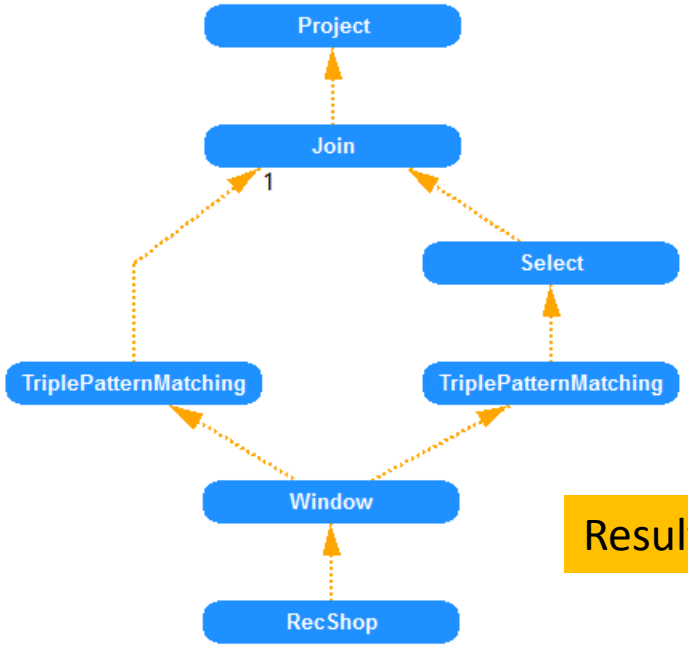
Resulting Query Execution Plan

# Filter

```
#PARSER SPARQL
#ADDQUERY
PREFIX cd: <http://www.recshop.fake/cd#>
SELECT ?x ?y ?price
FROM STREAM <System.recShop> WINDOW RANGE 10 MS ADVANCE 10 MS
WHERE {{?x cd:price ?price FILTER(toDouble(?price) > 10)}.
       {?x cd:artist ?y} }
```

Additional Brackets needed

Slightly modified filter (Odysseus needs to know the type of price at compile time)




Additional Filter

Resulting Query Execution Plan

# Aggregate Functions

```
#PARSER SPARQL
#ADDQUERY
PREFIX cd: <http://www.recshop.fake/cd#>
SELECT COUNT(?price)
FROM STREAM <System.recShop> WINDOW RANGE 10 MS ADVANCE 10 MS
WHERE {{?x cd:price ?price}.
       {?x cd:artist ?y} }
```

Currently, AVG, SUM, COUNT, MIN, MAX supported in SPARQL

# Odysseus

the event processing system

## Architecture

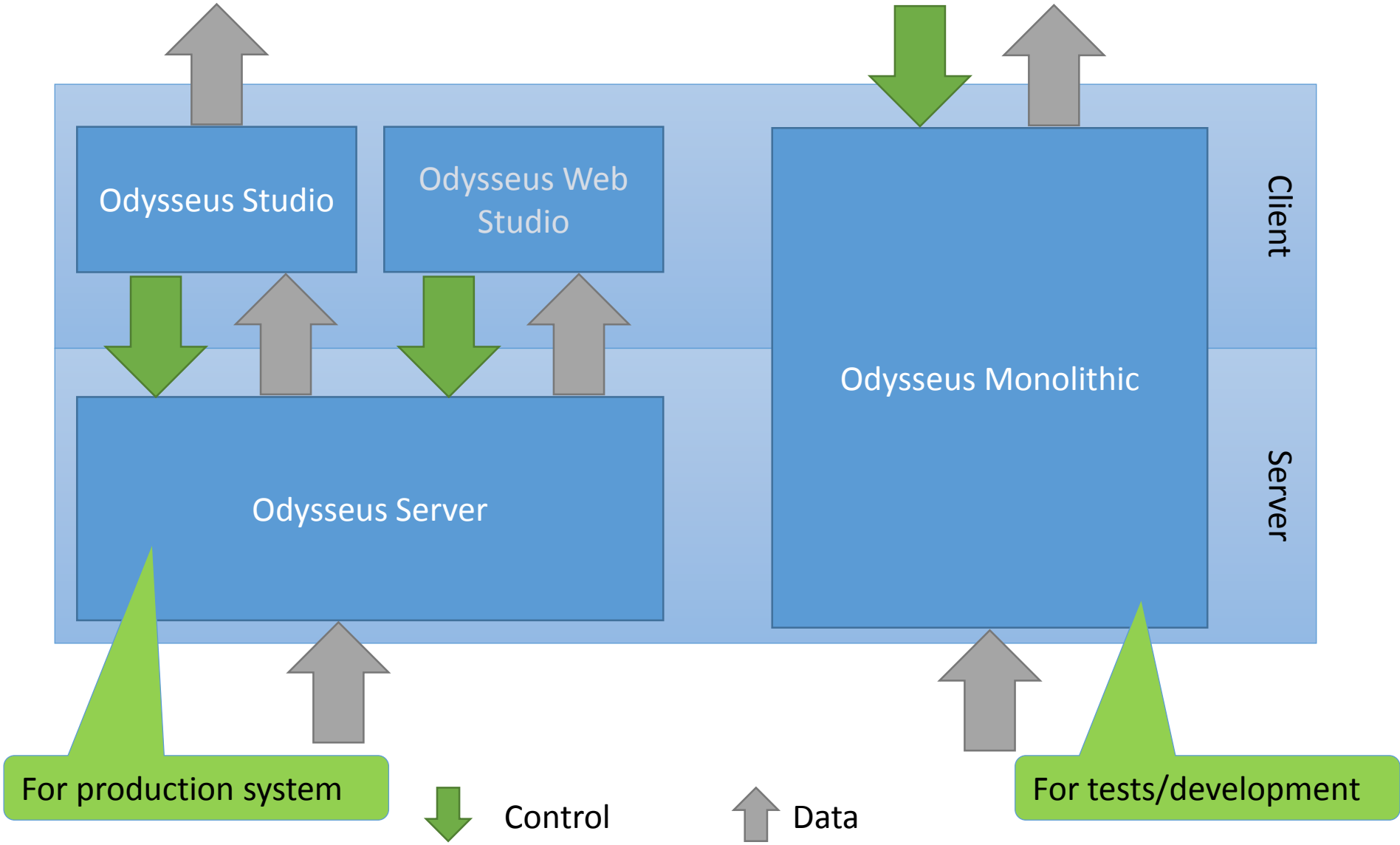
# Extensibility

- OSGi-based plug in system
  - Extensible and updatable at runtime
  - Plug-ins form a feature
  - feature free combinable

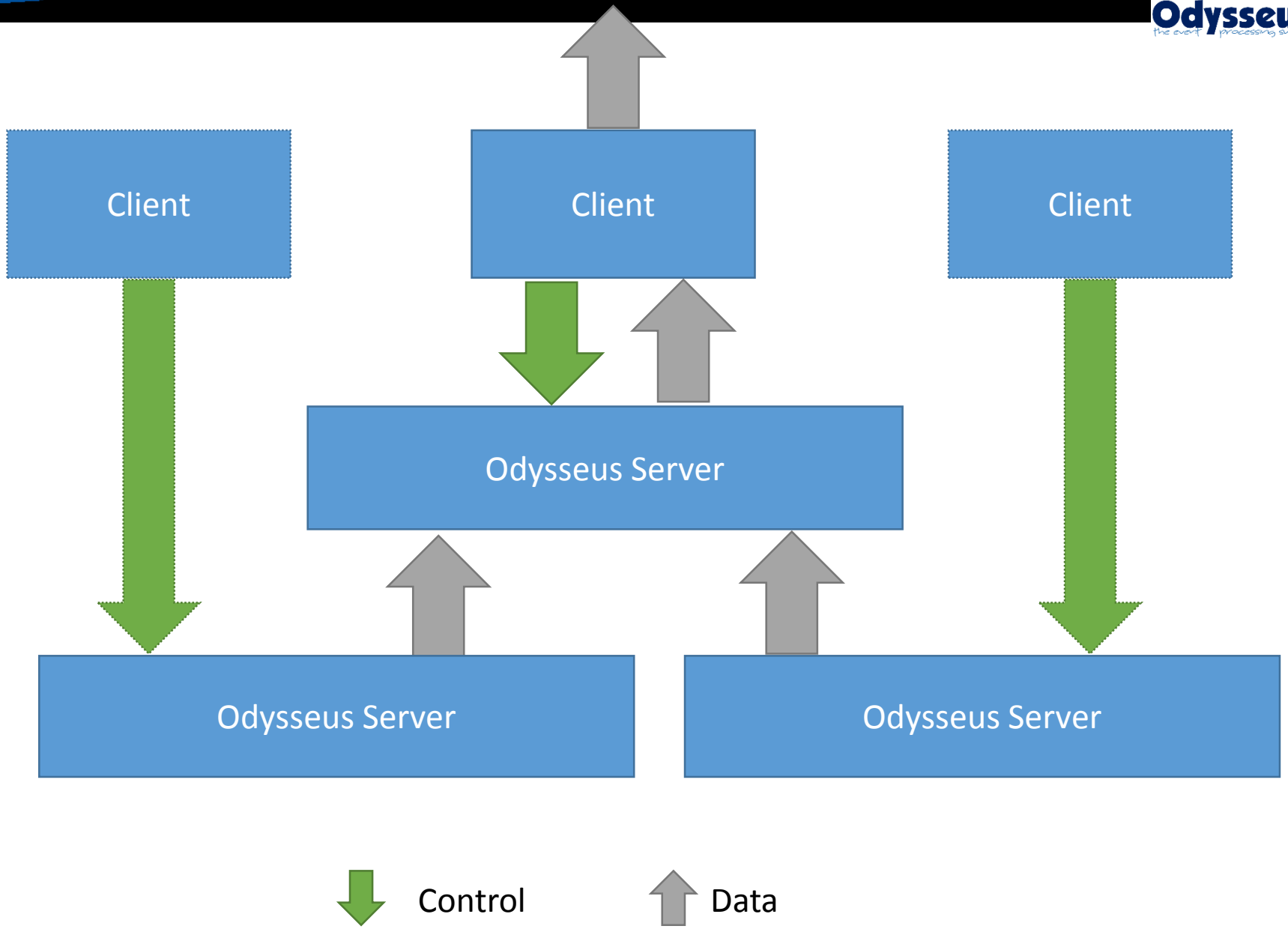
- Core features,:
  - Odysseus Core: common basis functions (Interfaces)
  - Odysseus Server: server/processing based functions (query processing, user management, web service, …)
  - Odysseus Studio: client based functions (RCP,…)

- Additional feature, can be installed, e.g.,
  - CEP: Complex pattern matching
  - Machine Learning: Clustering, classification, association analys
  - Spatial: Geo types and processing
  - RDF: Triple and SPARQL

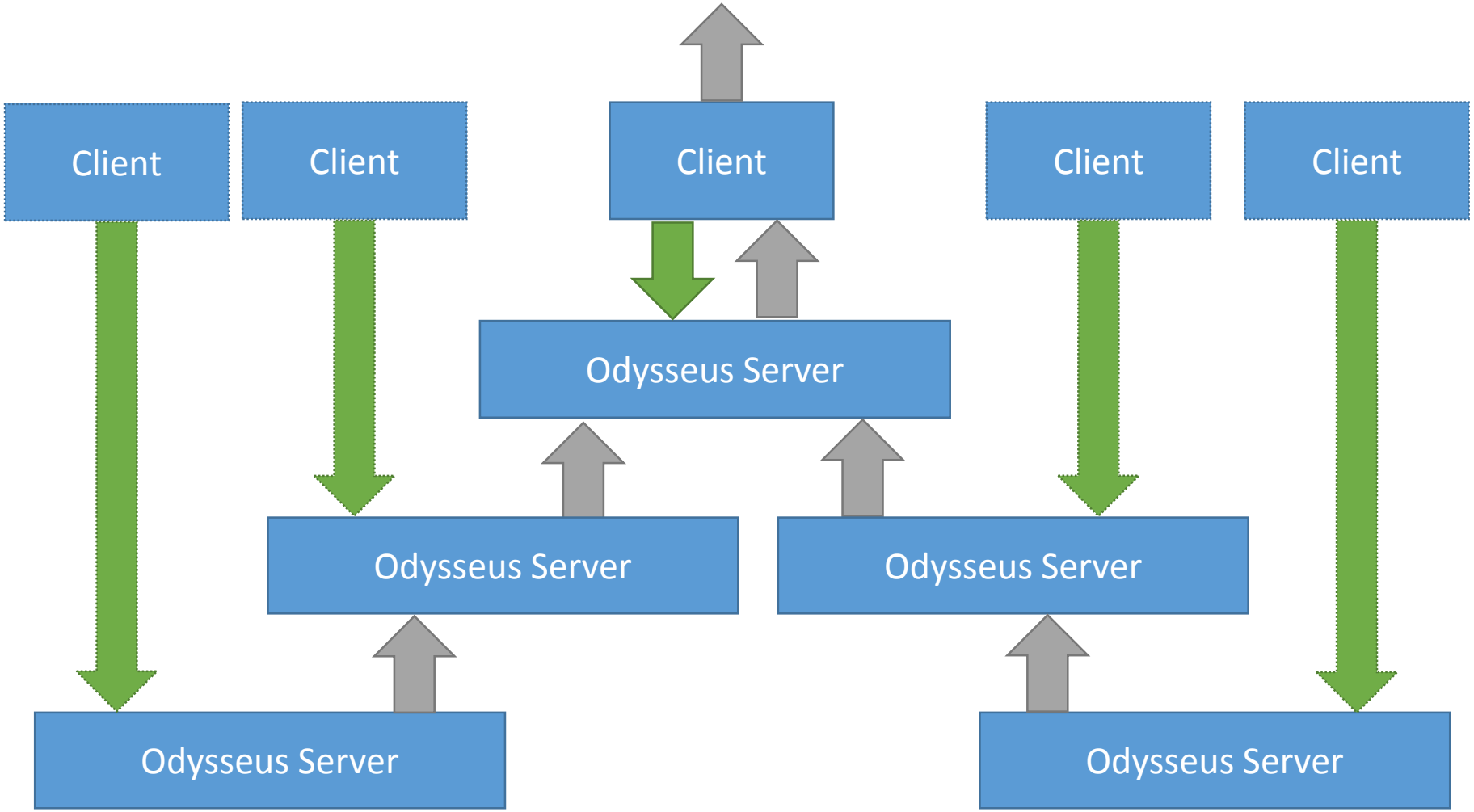
# Odysseus Architecture: Two versions



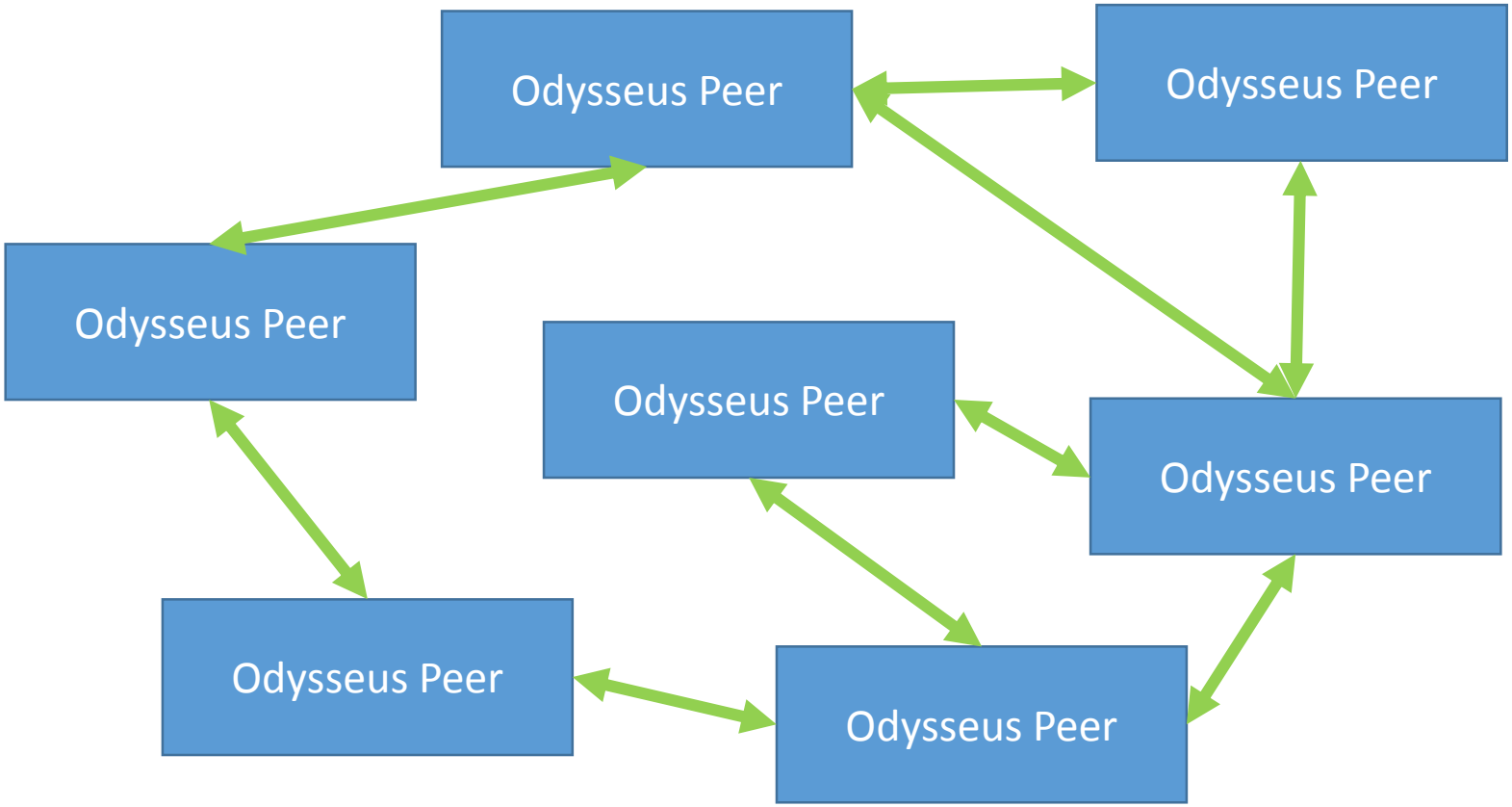
# Odysseus Architecture: Hierarchical



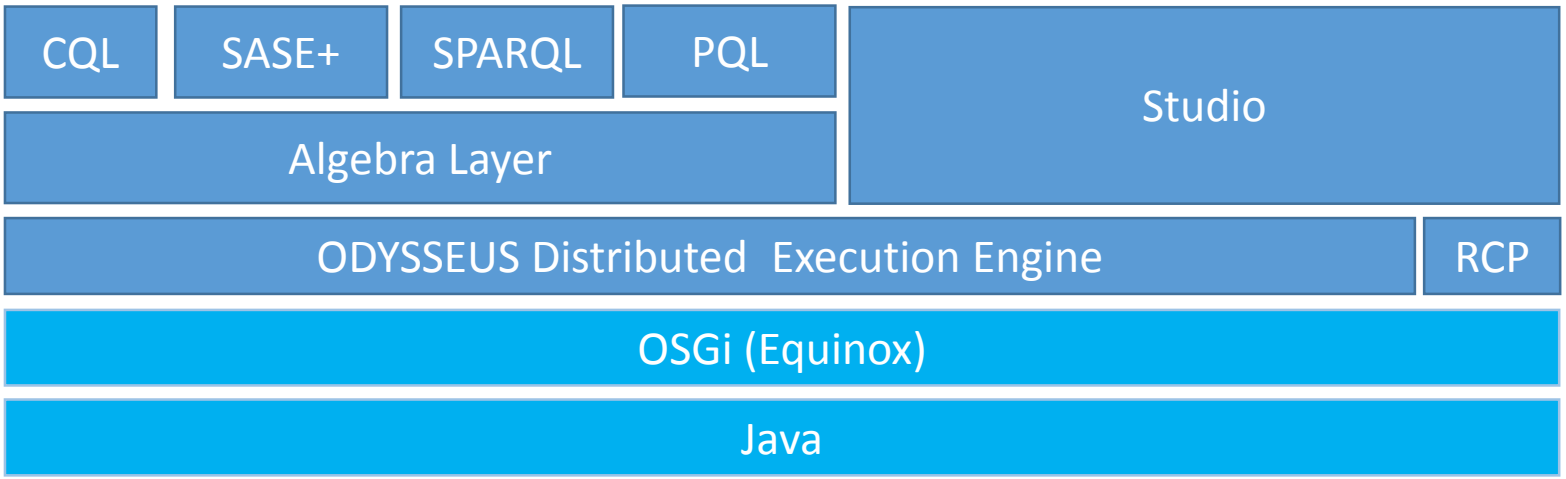
# Odysseus Architecture: Hierarchical



# Newest Version: Peer 2 Peer

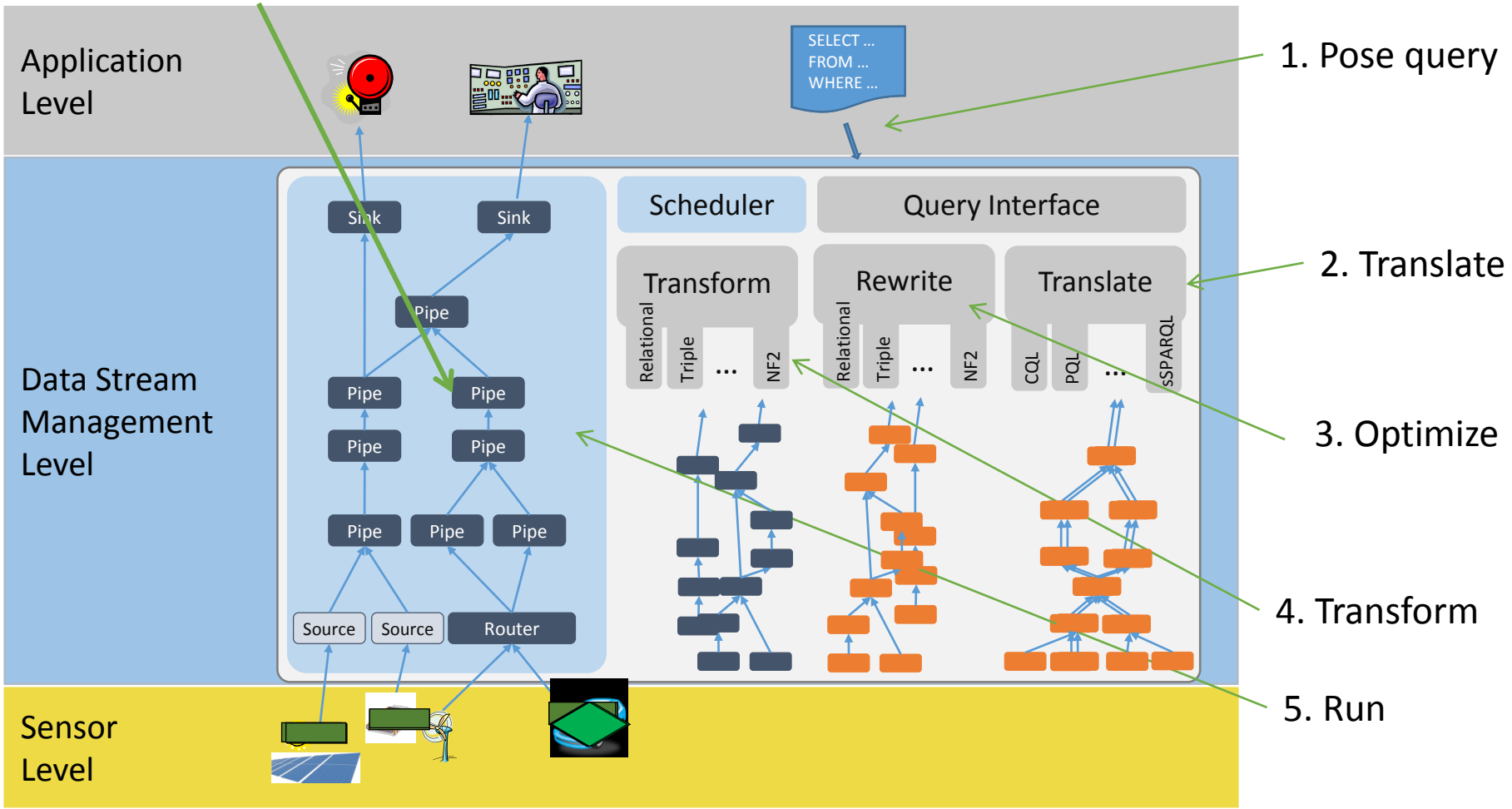


# Odysseus Software Stack



# Odysseus: Query procesing and optimization

Filter-Operator (Selektion)



# Odysseus

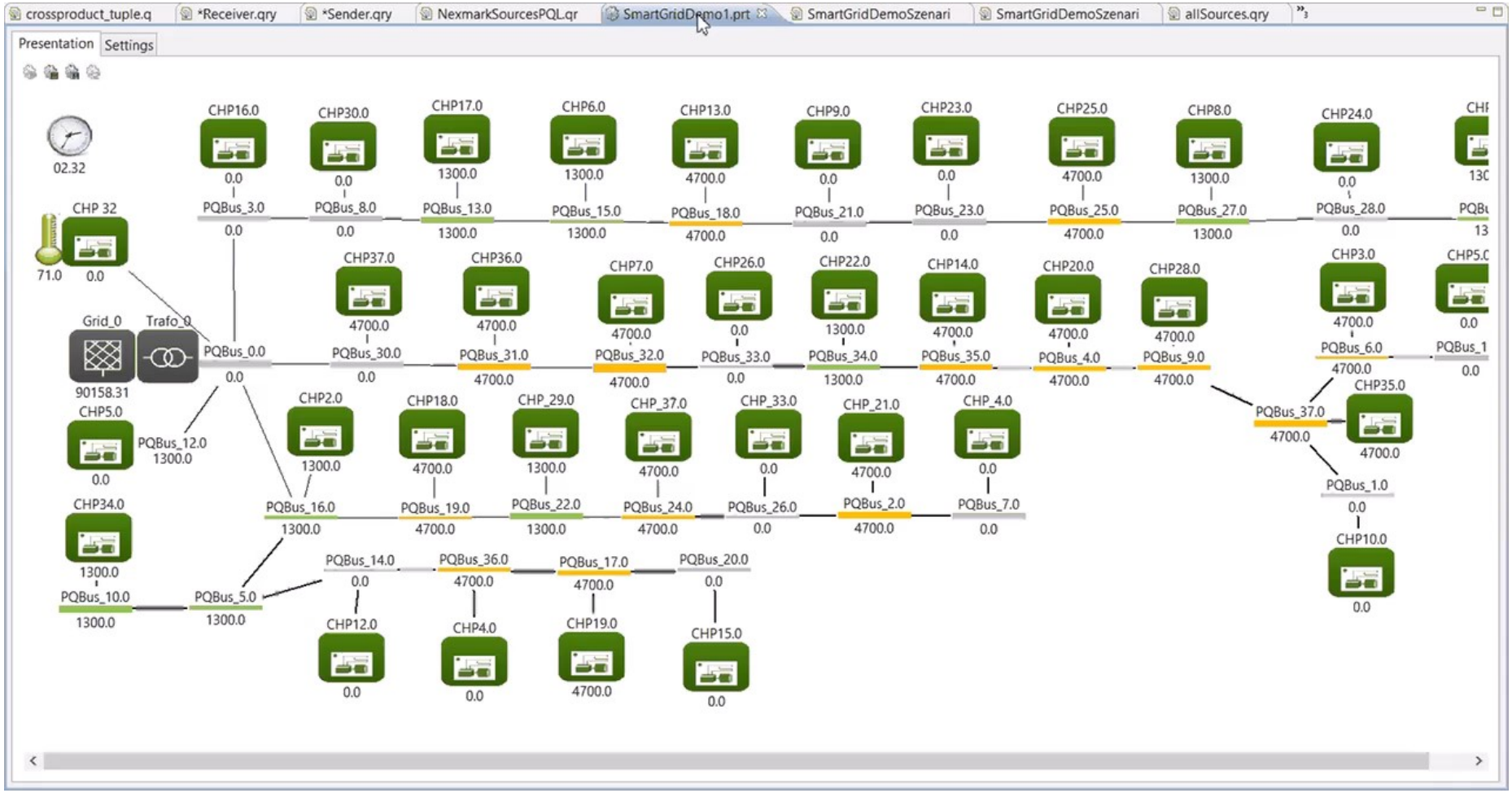
the event processing system

## Examples

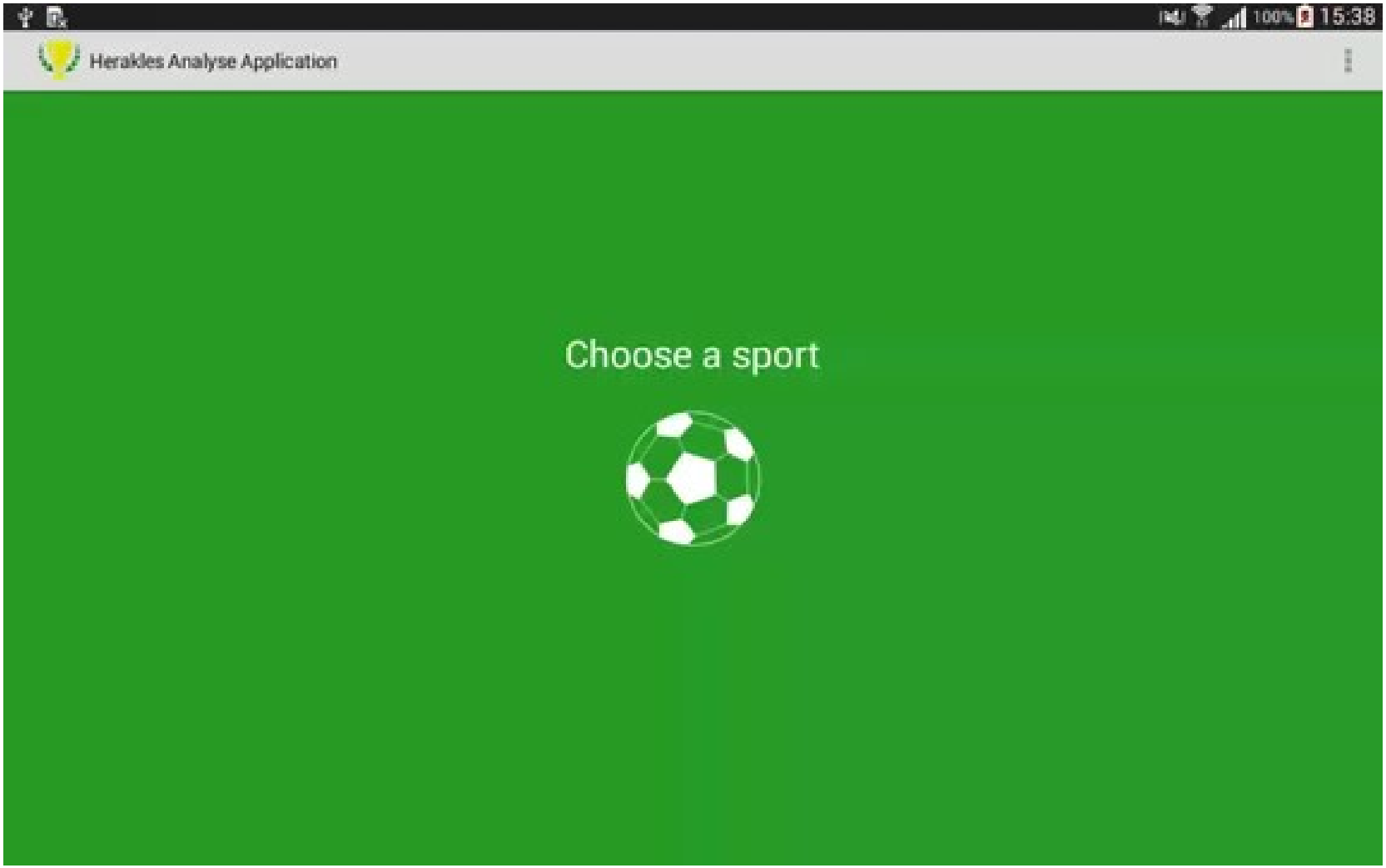
# Sample projects with Odysseus

- Scenarios, e.g.:
  - DEBS Grand Challenge 2012 (Manufacturing), 2013 (Soccer game), 2014 (SmartGrid), 2015 (Taxis in New York)
  - Many student work (e.g., Baggage Monitoring, SCADA, Condition Monitoring)
- OFFIS-Projects with Odysseus (Transportation), e.g.:
  - SOOP (Industry partner: Frisia Offshore, IBM): Support of Off-Shore Operations
  - COSINUS (Industry partner: Signalis, Raytheon Anschütz): Cooperative Shipping and Navigation on Sea
  - SALSA (Industry partner: Götting KG): Safe autonomic logistic and transportation in the outside area
  - eMIR/CSE (Signalis, Raytheon Anschütz, Atlas Elektronik): eMaritime Integrated Reference Plattform
  - SCAMPI (mercatis, akquinet, innotec, Müller (Drogeriekette)): Sensor Configuration and Aggregation Middleware for Multi Platform Interchange
- Geplant in OFFIS E
  - Data processing of a wind parks with Odysseus
- Further industrial applications
  - Monitoring of PV appliances
  - Monitoring in Industrial 4.0 scenarios

# Film: Smart Grid Monitoring



# PG: Lifeanalyse im Sport



# Current work in progress

- Better multi core utilization, currently inter query, new intra query:
  - Inter operator
  - Intra operator
- A framework for condition monitoring
- Main memory compression for „Big Windows“
- A new language for
  - Query definition
  - Operator definition (less work for new operators)
- A compilation framework
  - Create a query in Odysseus and translate to different targets
  - For scenarios where there is no Java/OSGi available

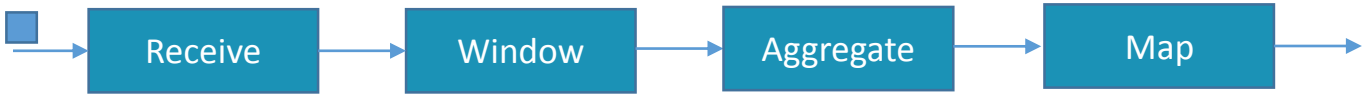
# Time for questions and discussion

Quelle: http://www.entspannt-lernen.de/fragen-antworten.html

Odysseus
the event processing system

# BACKUP

# Example with data stream

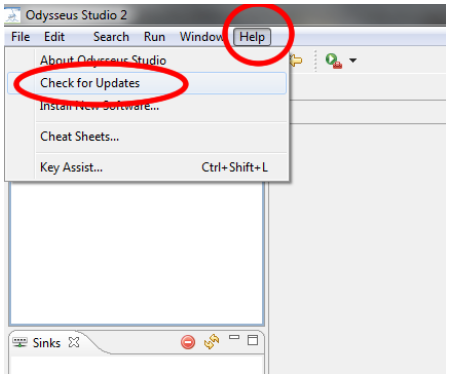


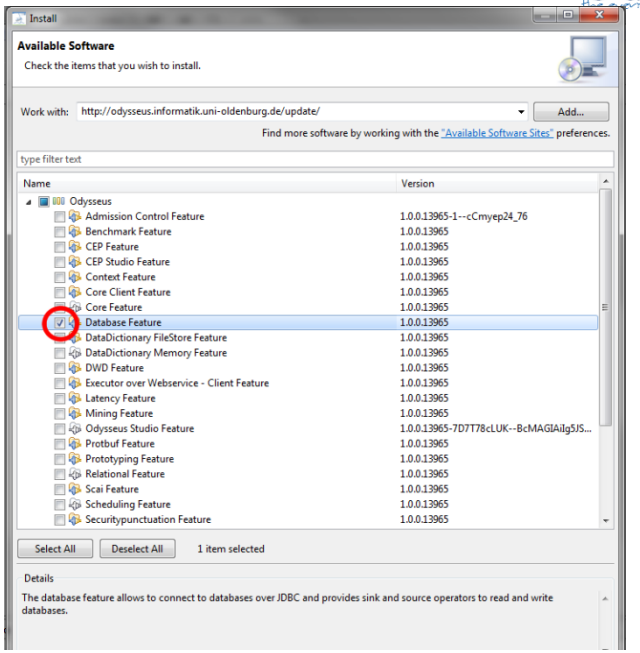

| | | | | |
|---|---|---|---|---|
| 1-100-... | (1,100,...) | (1,100,...) | (1,100,100) | (1,0) |
| 2-200-... | (2,200,...) | (2,200,...) | (2,100,**200**) | (2,**100**) |
| 3-150-... | (3,150,...) | (3,150,...) | (3,100,200) | (3,100) |
| 4-210-... | (4,210,...) | (4,210,...) | (4,100,**210**) | (4,**110**) |
| 5-90-... | (5,90,...) | (5,90,...) | (5,**90**,210) | (5,**120**) |

| WEA | |
|---|---|
| **PK** | **id** |
| | timestamp<br>load<br>location |

# Bereitstellung von Features

- Installation, Update über zentrales Repository




- Oder über Konsole

```
.usermanagement.AbstractUserManagement.initDefaultUsers(AbstractUserManagement.java:656)
2728 DEBUG AbstractUserManagement  - Creating new user database for Tenant Marco3 - de.uniol.inf.is.odysseus.core.server
.usermanagement.AbstractUserManagement.initDefaultUsers(AbstractUserManagement.java:656)


osgi> installedFeatures
21636 INFO  FeatureUpdateUtility  - Starting task ... - de.uniol.inf.is.odysseus.updater.FeatureUpdateUtility$4.beginTas
k(FeatureUpdateUtility.java:432)
21644 INFO  FeatureUpdateUtility  - 1% completed - de.uniol.inf.is.odysseus.updater.FeatureUpdateUtility$4.worked(Featur
eUpdateUtility.java:380)
21645 INFO  FeatureUpdateUtility  - 100% completed - de.uniol.inf.is.odysseus.updater.FeatureUpdateUtility$4.done(Featur
eUpdateUtility.java:419)
21646 INFO  FeatureUpdateUtility  - Task  done - de.uniol.inf.is.odysseus.updater.FeatureUpdateUtility$4.done(FeatureUpd
ateUtility.java:423)
Following features are installed:
 - de.uniol.inf.is.odysseus.common.feature.feature.group
 - de.uniol.inf.is.odysseus.core.server.feature.feature.group
 - de.uniol.inf.is.odysseus.planmgmt.standard.feature.feature.group
 - de.uniol.inf.is.odysseus.relational.feature.feature.group
 - de.uniol.inf.is.odysseus.scheduling.feature.feature.group
 - de.uniol.inf.is.odysseus.script.feature.feature.group
 - de.uniol.inf.is.odysseus.server.feature.feature.group
 - de.uniol.inf.is.odysseus.server.platform.feature.feature.group
 - de.uniol.inf.is.odysseus.usermanagement.filestore.feature.feature.group
 - org.eclipse.equinox.p2.core.feature.feature.group

osgi>
```

# Odysseus

the event processing system

## Qualitätssicherung

# Qualitätssicherung

- Versionskontrolle: Zentrales SVN mit Zugriffsbeschränkungen
- Continuous Integration
- Erweiterbares Framework für Integrationstest
  - Ausführen von Testanfragen
  - Abgleich mit erwarteten Anfrageergebnissen

- Jenkins
  - Stündlich: Überprüfung auf Compilefehler, Integrationstests
  - Täglich: zusätzlich Produkte und Repository erzeugen

- Collaboration-Tools
  - Ticketsystem: JIRA
  - Dokumentation: Confluence
  - SourceCode: FishEye

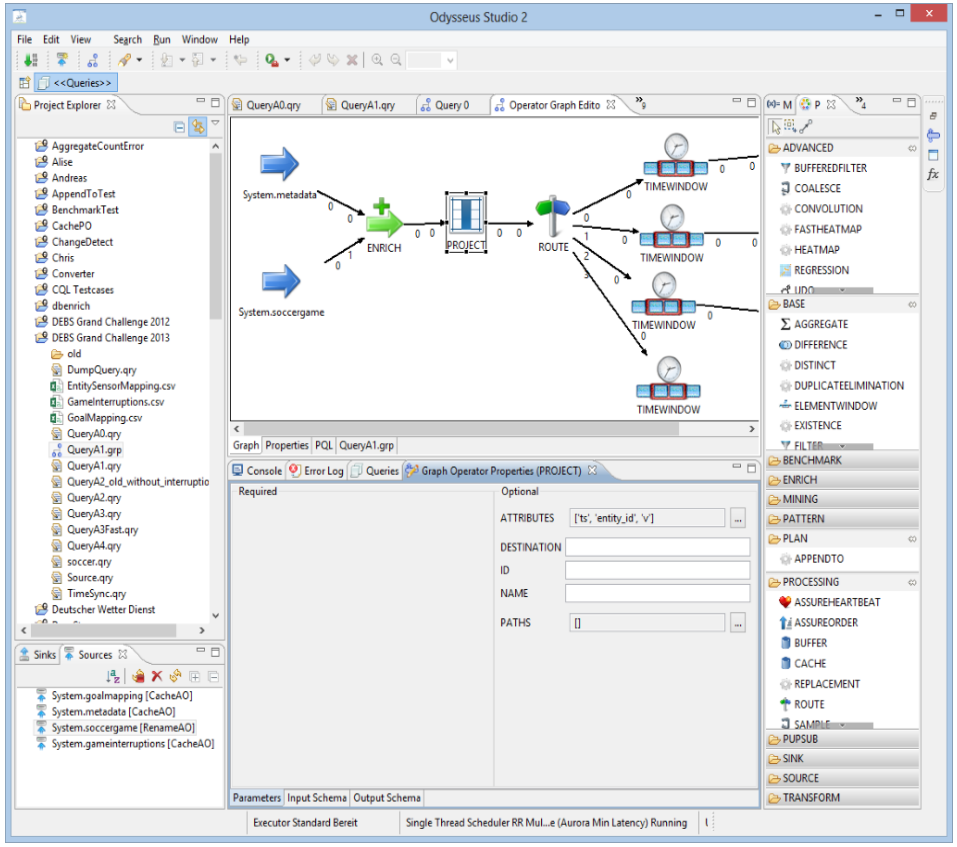
# Odysseus

the event processing system

## Odysseus Studio

# Odysseus Studio

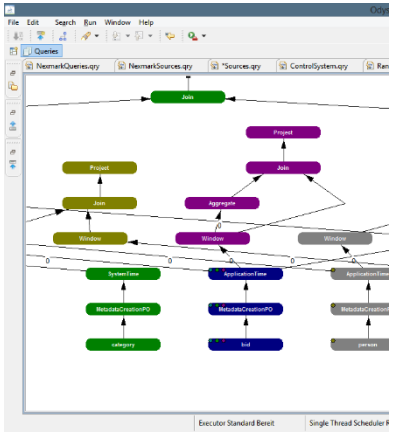
- RCP basierte Client-Anwendung zur
  - Überwachung von Odysseus
  - Verwalten von Anfragen
  - Visualisieren von Anfrageergebnissen



# Odysseus Studio

- Formulierung von Anfragen
  - Syntaxhighlighting, Autovervollständigung, …
- Visueller Anfrageeditor
  - Drag‘n‘Drop von Operatoren
- Anzeige von Ergebnissen
  - Texte, Diagramme, Karten, …
- Dashboard-Framework
  - Kombination von Diagrammen etc. mit zugehöriger Anfrage
  - Visualisierung von Ergebnissen
- In monolithischer Version
  - Visualisieren des Anfrageplans
  - Anzeige von Zwischenergebnissen im Plan
  - Debugging von Anfragen (durch Stoppen und Steppen)

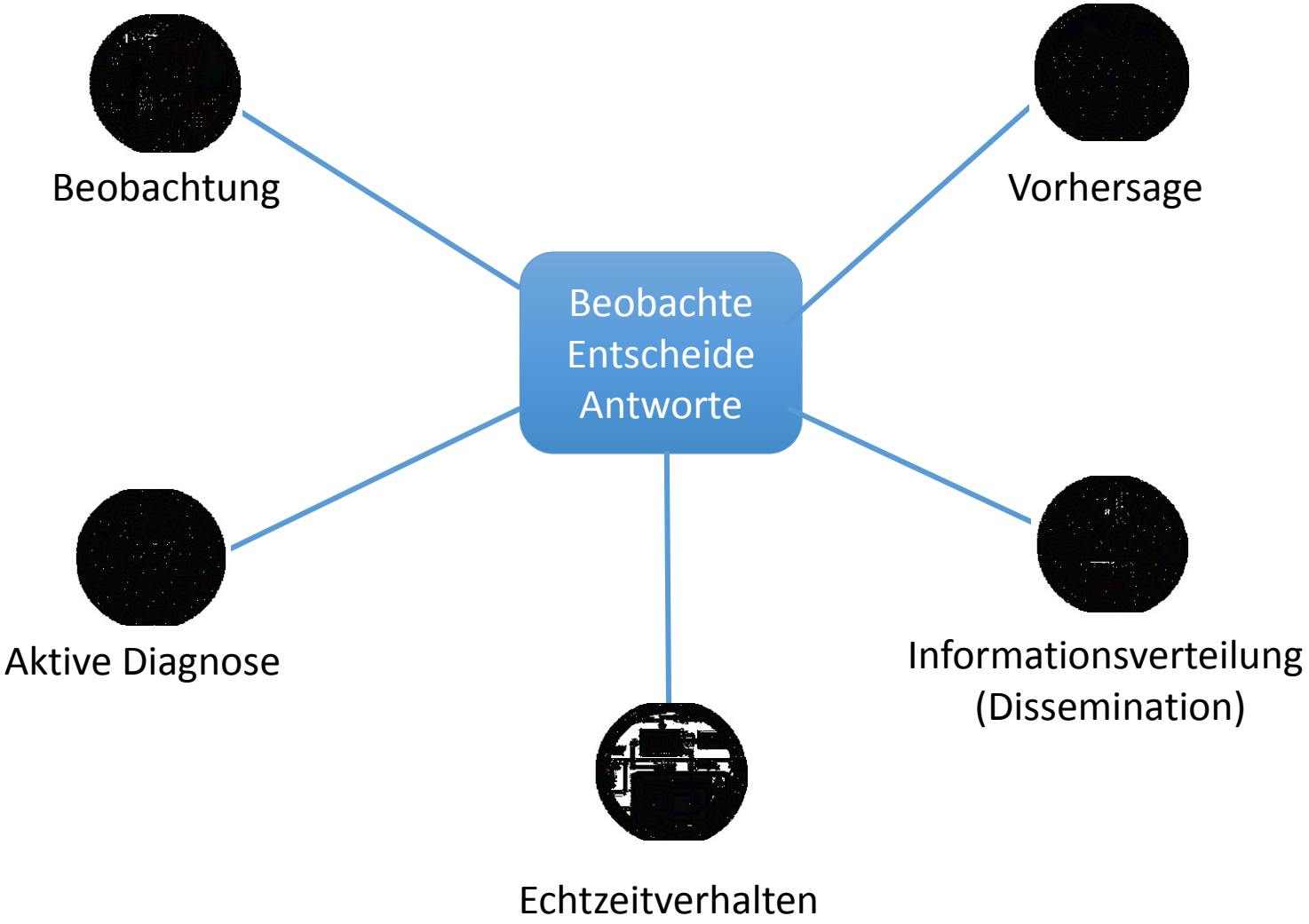
```
#SCHEDULER "ST Scheduler RR MS Limit Thr
#BUFFERPLACEMENT Query Buffer Placement
#DOREWRITE true
#DEFINE SIZE 600000
#DEFINE delay 1
#DEFINE TIMES 10
///#DEFINE maxlines 1000
#TRANSCFG Standard
#LOOP i 1 UPTO ${TIMES}
#PARSER PQL
#ADDQUERY
source${i} ::= TIMESTAMP({start='datetim
                    ACCESS({
                        source='nrel${i}
                        wrapper='Generic
                        transport='file'
                        protocol='simple
                        datahandler='tup
```



# Diskussionspunkte

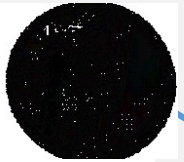
- Entwicklungsaufwand: „How long to get it run?“
  - Installation der Software < 1 h
  - Integration der Quellen → Je nach Komplexität < 1 Woche pro Quellentyp
  - Funktionalität → Anfragen → Je nach Komplexität, eher im Stunden oder Tagebereich (→ wenn man das Problem verstanden hat)
- Wartungsaufwand
  - laufender Betrieb: Vergleichbar mit Datenbanksystem
  - Änderungen: Abhängig von Komplexität
    - Neue Anfragen
    - Neue Funktionalitäten
    - Neue Adapter

# Gründe für den Einsatz von Ereignisverarbeitung



# Gründe für den Einsatz von Ereignisverarbeitung

Schnelle Erkennung außergewöhnlichen (Geschäfts-)Verhaltens und Benachrichtigung entsprechender Personen → Alarmfunktion



**Beobachtung**

typische Beispiele:

- Überwachung Energieerzeugung
- Patientenüberwachungssystem
- Gepäckabfertigung
- Fabrikgelände

Beobachte
Entscheide
Antworte

# Gründe für den Einsatz von Ereignisverarbeitung

Liefere die richtige Information zum richtigen Empfänger in der richtigen Granularität zum richtigen Zeitpunkt → Personalisierte Informationszustellung

**Informationsverteilung**

typische Beispiele:

- Notfallsteuerungssystem

# Gründe für den Einsatz von Ereignisverarbeitung

Beobachte
Entscheide
Antworte

Unmittelbare Reaktion (mehr als Nachrichten)
auf eingehende Ereignisse

**Echtzeitverhalten**

typische Beispiele:
- Energieerzeugung
- Handelssystem
- Maut

# Gründe für den Einsatz von Ereignisverarbeitung

Beobachte
Entscheide
Antworte

Erkenne ein Problem anhand von beobachteten Symptomen

**Aktive Diagnose**

typische Beispiele:
- Energieerzeugung
- Produktionssteuerung
- Helpdesk

# Gründe für den Einsatz von Ereignisverarbeitung

Entschärfe oder vermeide vorhergesagte Ereignisse

**Vorhersage**

Beobachte
Entscheide
Antworte

Beispiel:

- Finanzüberwachung
- Geräteüberwachung

# Warum Odysseus

- Effiziente hauptspeicherbasierte Verarbeitung auf Basis etablierter Datenbanktechnologien
- Verwendung von Anfragesprachen anstelle von komplexen Quellcodes
- Semantisch einheitliche und reproduzierbare Verarbeitung von Events, u.a. anhand von Zeitstempel
- Einfache Integration in bestehende Umgebungen durch anpassbare Adapter und Webservice-Technologien
- Flexibel, anpassbar und erweiterbar durch komponentenbasierte Architektur
- Mehrbenutzerfähigkeit und Client-Server-Fähigkeit
- Betriebssystemunabhängig durch Java (u.a. Raspberry Pi)