Distributed Data Management

Summer Semester 2013

TU Kaiserslautern

Dr.-Ing. Sebastian Michel

smichel@mmci.uni-saarland.de

Lecture 10

DISTRIBUTED DATA STREAM PROCESSING / SENSOR NETWORKS

Recap Data Stream Management

- Previous lecture: generic concepts of **sliding windows** and **continuous queries**; semantics of continuous query language (CQL) for data stream processing

Sample Query:

```
SELECT F.clerk, max(O.cost)
FROM orders O,
     fulfillments F [PARTITION BY clerk ROW 5] 10% SAMPLE
WHERE O.orderID = F.orderID
GROUP BY F.clerk
```

Content of Today's Lecture (Cont'd)

- Implementation/query processing concepts

- Distributed DSMS with emphasis on recent systems like Twitter Storm or Yahoo S4 that aim at fault tolerant large-scale processing

- Sensor networks. Particularly, managing sensor data using sensor data middleware.

Query Processing

- Many problems to be addressed resemble conceptually the same issues that arise in traditional RDBMS
- Goals of DSMS as different in many aspects, though.
  - Continuous queries
  - Push-based data model
  - Aim at real-time processing
  - Need for memory efficient algorithms
  - Handle overload to guarantee real-time processing; load shedding
  - Sharing of intermediate results (multi query optimization)

Implementation and Processing

- Query is compiled into query execution plan (similar to what is known from RDBMS lectures)

- Recall differences from DBMS and DSMS; data is actively streaming in.

- What does this imply for the implementation?

Push vs. Pull

- Two fundamentally different ways operators (nodes in a query plan) interact

- **Pull:** Consuming operator actively retrieves results of producer.
- **Push:** Producer push results to consumer.

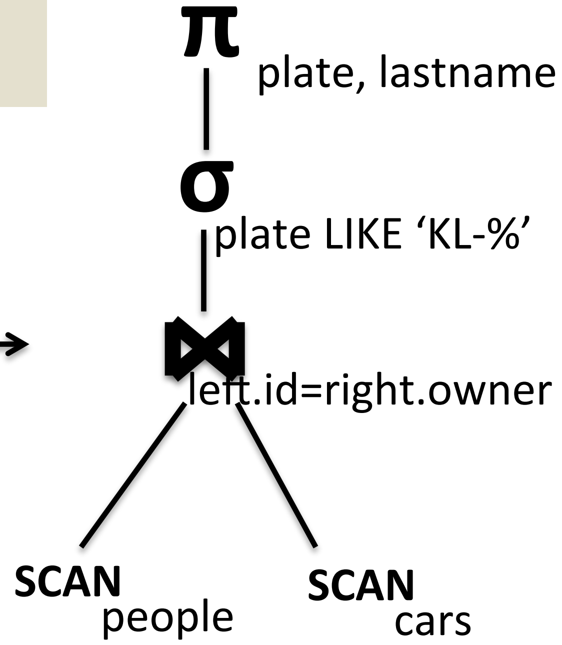
Pull

- We all know that from DBMS (think JDBC or operator trees) or Java Iterators

```
ResultSet rset = Statement.executeQuery(“Select * from ….”);
while (rset.next()) {
        rset.getInteger(1);
        …
}
```

```
SELECT c.plate, p.lastname
FROM people p JOIN cars c ON p.id=c.owner
WHERE c.plate LIKE ‘KL-%’
```

→



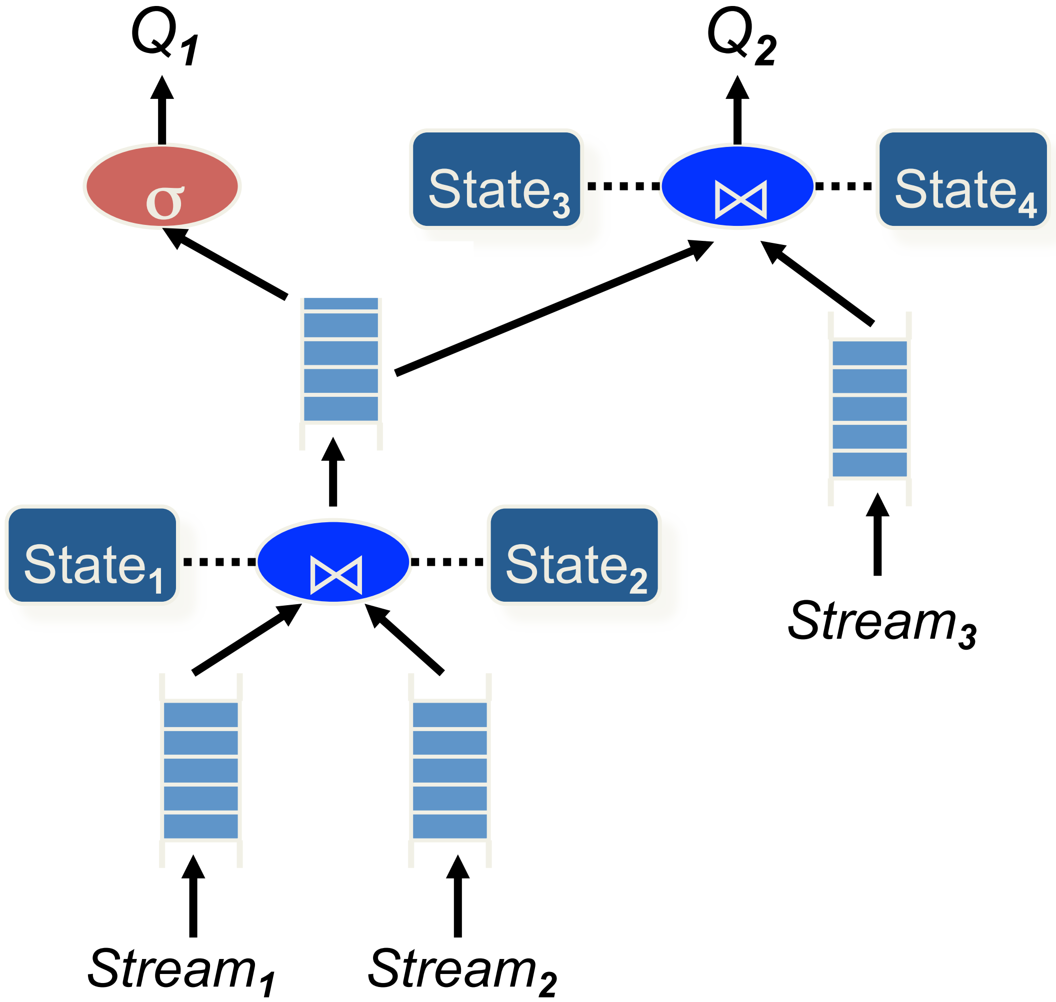
“OPEN, NEXT, CLOSE”

Push

- Steam processing is by design mainly data-driven
- Operators register at other operators
- When new tuples are generated, they are actively pushed to registered operator

- Creating a directed acyclic graph (DAG), e.g., called topology in later system

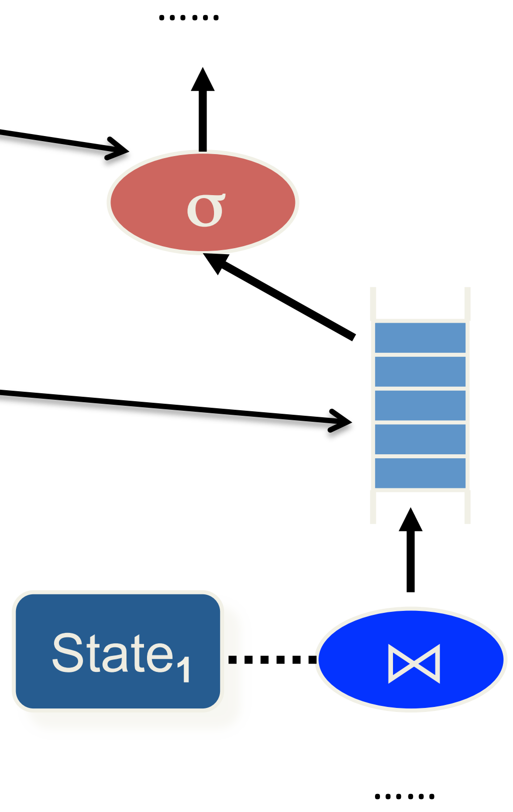
STREAM: Simple Query Plan



Slide courtesy of Jennifer Widom.

Query Plans in STREAM

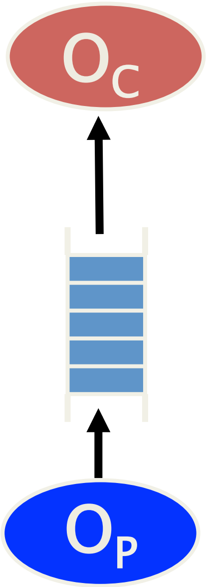
- Operators
  - do the actual processing;
  - e.g., join, selection, window, …
- Queues
  - connect operators
- Synopses
  - store operator states. For instance, the hash table of a hash-based join



Queues

- A queue connects a tuple producing operator O_P and its consuming operator O_C
- Conceptually FIFO buffer
- Elements inserted and retrieved in timestamp order

- Shared Queues: multiple consumers for one producer possible



Operator Decoupling

- Queues allow decoupling of operators
- Consumers read from queue
- Producers write to queue

Distributed DSMS

- Conceptually distributed data stream management systems behave/look like centralized ones

- STREAM (seen before)
- Borealis (Brandeis U, Brown U, MIT)
- Global Sensor Networks (EPFL)
- …

Abadi et al. : The Design of the Borealis Stream Processing Engine. CIDR 2005: 277-289

Karl Aberer et al.: Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. MDM 2007: 198-205

Distributed DSMS (Cont’d)

- In spirit of the beginning of the lecture on MapReduce / NoSQL, we look at very recent distributed DSMS for big data (stream) processing
  - Yahoo! S4 (now Apache)
  - Twitter Storm
- Many concepts are also generic. Conceptually, e.g., the operator interfaces and topologies.

(Generic) Aims

- Guaranteed data processing
- Fault tolerance
- Horizontal scalability
- Enable high-level programming

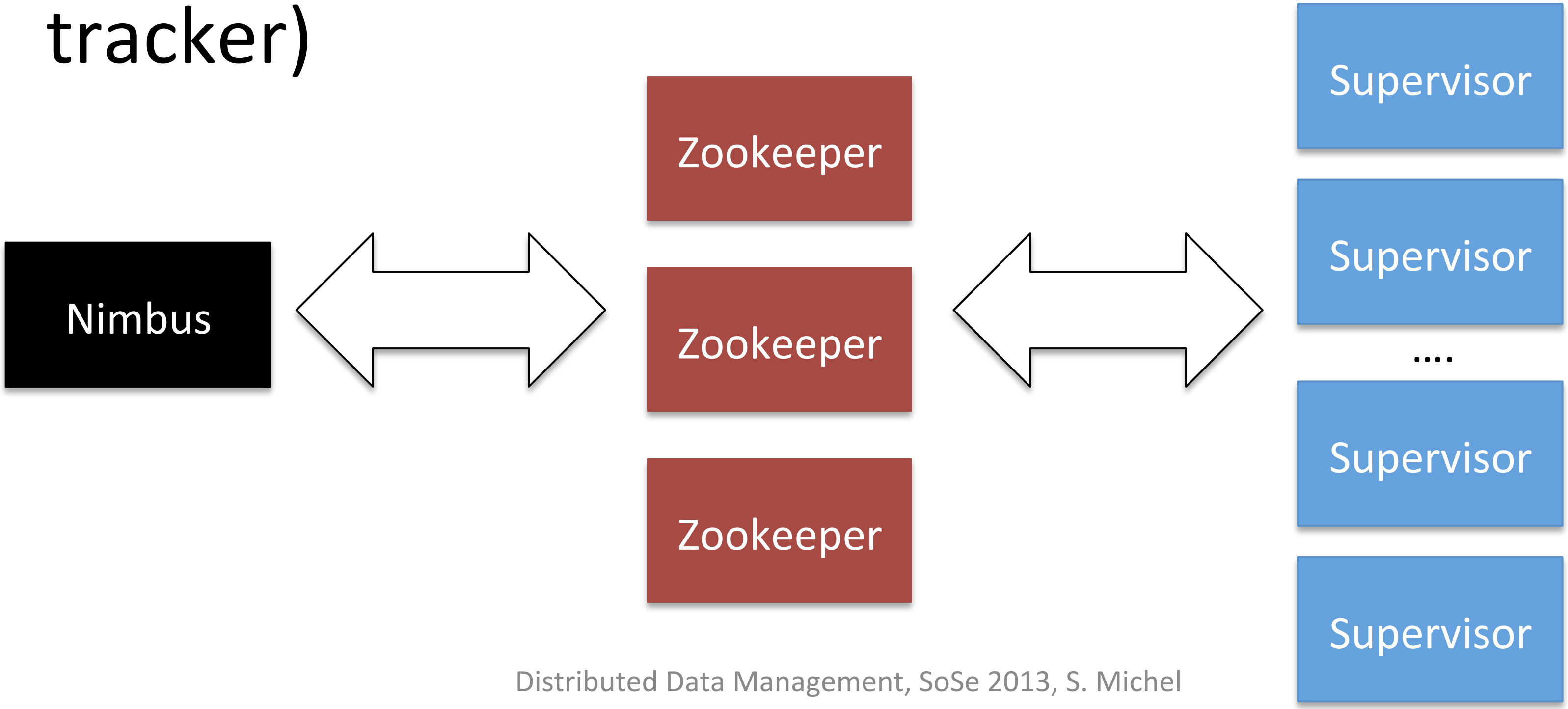
- Sounds like MapReduce/Hadoop? Well …

Twitter Storm

- Sometimes referred to as "the realtime Hadoop"
- Fault tolerant, distributed stream processing system. Developed by N. Marz (now Twitter) in 2011
- Widely used by companies
- Data stream operators are (can) be put on different nodes; replicated operators of same kind for scalability.

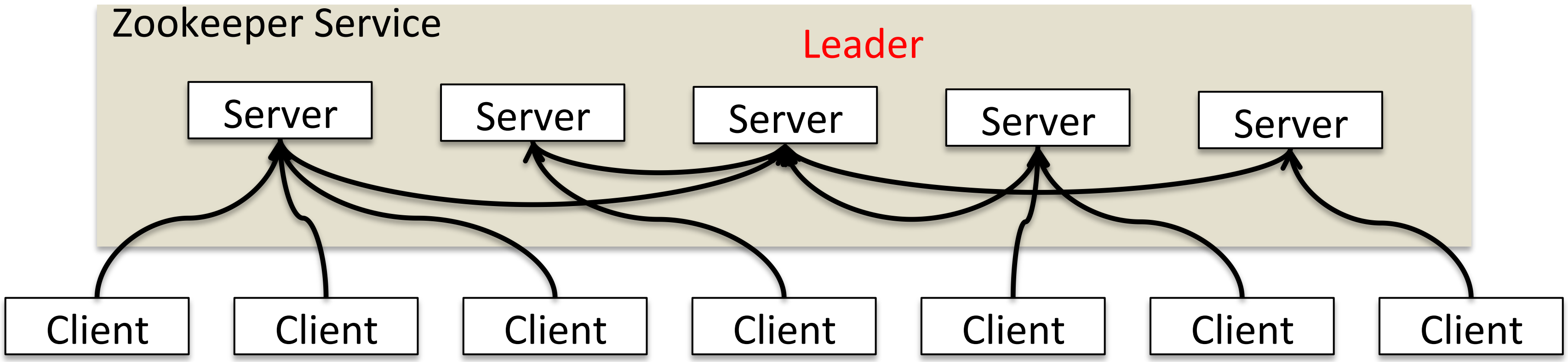
Storm Cluster Setup

- Using Apache Zookeeper for coordination
- Supervisor: worker nodes (like Hadoop task tracker)
- Nimbus: coordinator node (like Hadoop job tracker)



Zookeeper: Setup + Data Model

- “enables highly reliable distributed coordination”



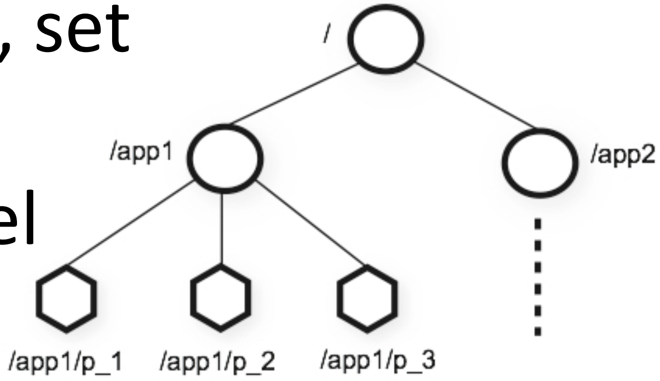
Zookeeper Service

Leader

Server Server Server Server Server

Client Client Client Client Client Client Client

- Hierarchical data model, simple API: create, delete, exists, get data, set data, get children, sync
- Used to implement higher level applications



http://zookeeper.apache.org/doc/trunk/zookeeperOver.html

Zookeeper Guarantees

- Sequential Consistency: Updates from a client will be applied in the order that they were sent.
- Atomicity: Updates either succeed or fail.
- Single System Image: A client will see the same view of the service regardless of the server that it connects to.
- Reliability: Once an update has been applied, it will persist from that time forward until a client overwrites the update.
- Timeliness: The clients view of the system is guaranteed to be up-to-date within a certain time bound.

Storm and Zookeeper

- Storms use Zookeeper for
  - Discovery of nodes
  - Storing state of nimbus and supervisors
  - Guaranteed message processing/tracking
  - and storing statistics

- The actual heavy communication between nodes is using a library called Zero MQ

http://www.zeromq.org/

Zookeeper Application

- Barrier: synchronize beginning and end of computation for group of processes
- Enter:
  - zk.create(root + “/" + myProcessName)

- Leave: while true
  - List<String> list = zk.getChildren(root, true);
  - break/return if list.size()==0, otherwise wait
- Or implementation of producer/consumer queues or distributed locks

read on: http://zookeeper.apache.org/doc/r3.2.2/zookeeperTutorial.html

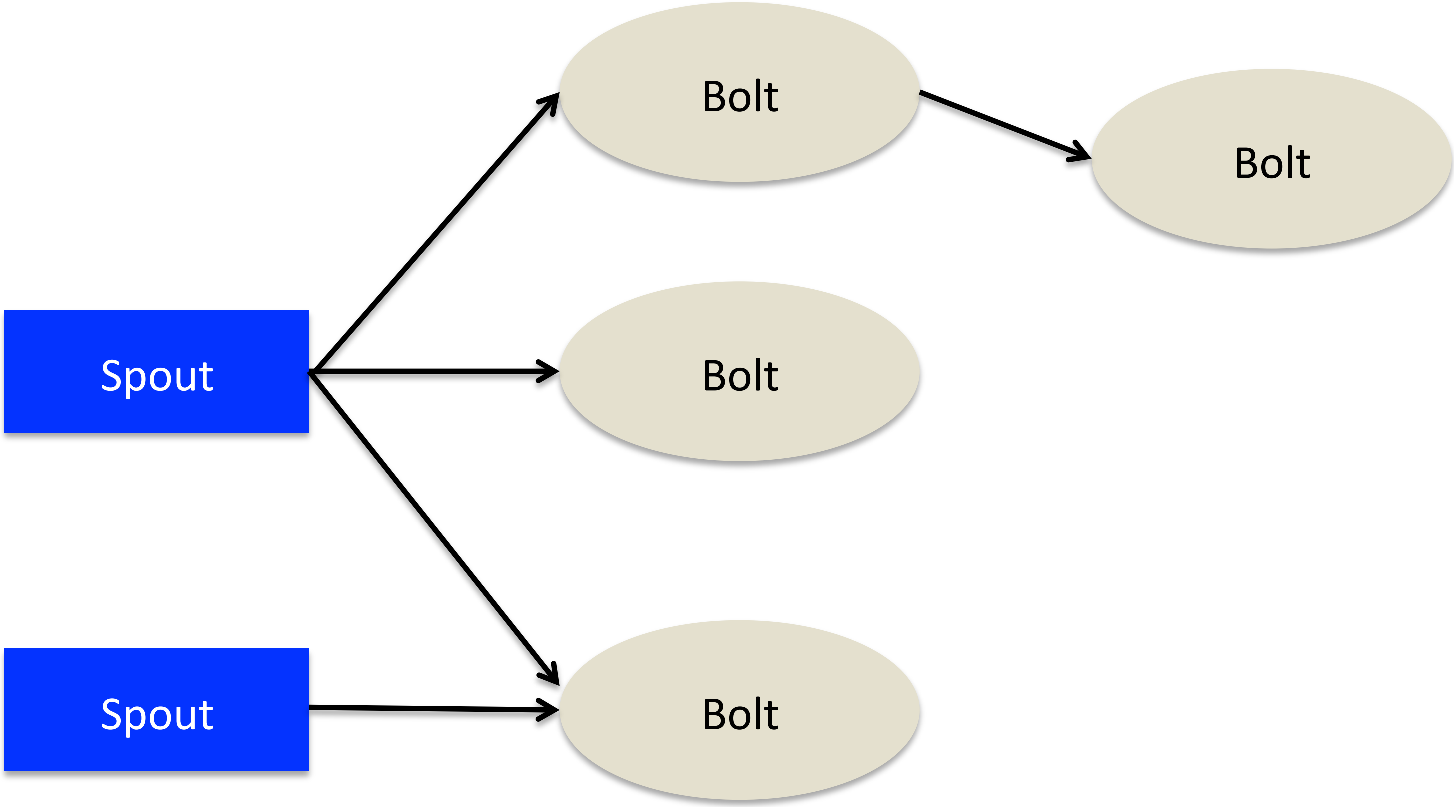
Storm Concepts

Data sources, operators and query plans are called in Storm:

- Spouts: Data sources (e.g., Twitter stream)
- Bolts: Operators that consume output of spouts or other bolts (e.g., filter stopwords)
- Topologies: By connecting spouts and bolts, the created topology determines the data flow.

Bolts and Spouts: Topology

- **Example Topology**

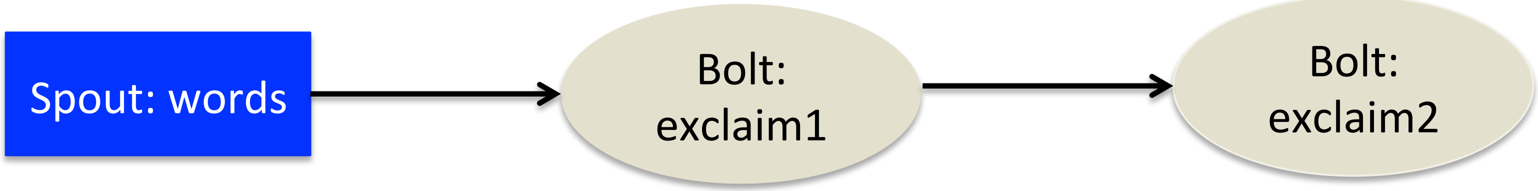


Topology Builder

- Operator/Source Topology is created by registering consumers to producers using unique names of sources/operators.

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
        .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
        .shuffleGrouping("exclaim1");
```



https://github.com/nathanmarz/storm/wiki/Tutorial

Topology Builder: Spouts

name of stream

```
builder.setSpout("words",
    new TestWordSpout(), 10);
```

Java Object that implements the Spout interface

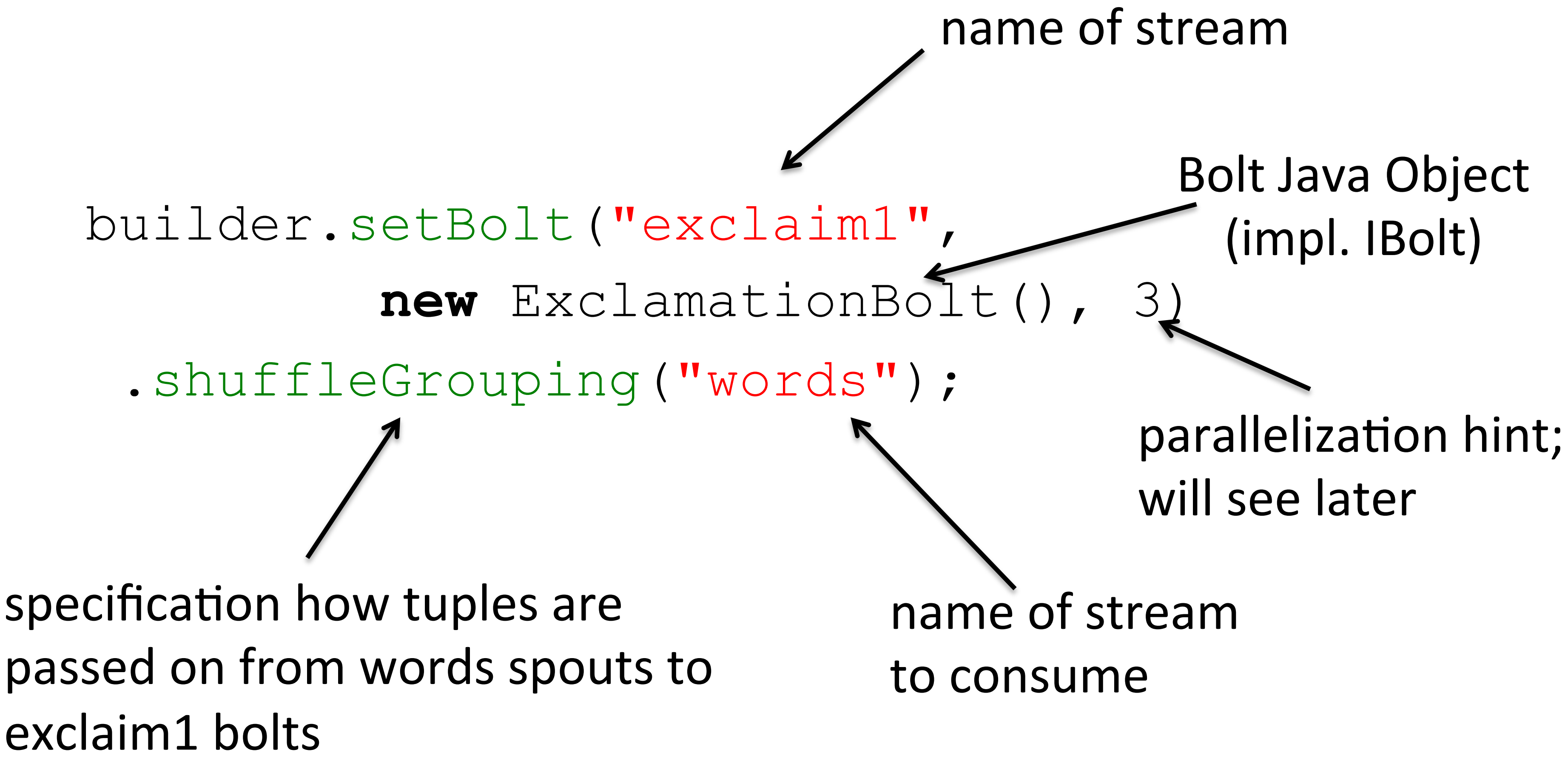
parallelization hint; will see later

Spout Implementation

- Sample Spout that emits at random words of a specific set

```
public void nextTuple() {
    Utils.sleep(100);
    final String[] words =
            new String[] {”a", ”b", ”c", ”d", ”e"};
    final Random rand = new Random();
    final String word =
                   words[rand.nextInt(words.length)];
    _collector.emit(new Values(word));
}
```

Topology Builder: Bolts



name of stream

Bolt Java Object
(impl. IBolt)

```
builder.setBolt("exclaim1",
          new ExclamationBolt(), 3)
 .shuffleGrouping("words");
```

parallelization hint;
will see later

specification how tuples are passed on from words spouts to exclaim1 bolts

name of stream to consume

Bolt Implementation

- Stream operators receive tuples and output (emit) new tuples

```
public void execute(Tuple tuple) {
       _collector.emit(tuple,
                new Values(tuple.getString(0) + "!!!"));
       _collector.ack(tuple);
}
```

- We will see later why tuples are acknowledged once processed.
- There are also a couple of other methods required, such as description of output fields

Stream Grouping Commands

- Shuffle Grouping: randomly spreading tuples across consuming operators. Good for load balancing.
- Fields Grouping: tuples are send to consumers based on specific fields.
- All Grouping: tuples are replicates among ALL of the bolts tasks
- Global Grouping: tuples go to ONE specific task (the one with lowest id)

- *There are some more groupings; see online description for full details (of current state)*

Example Application

- Counting the mentions of #hashtags in the Twitter stream.
- Counting how often two #hashtags co-occur.
- Computing trends as sudden increases of such occurrences or co-occurrences.
- Grouping (particularly by field) needs to make sure required data is arriving at nodes that do counting for a #hashtag or pair, e.g.

Application of Fields Grouping: Joins

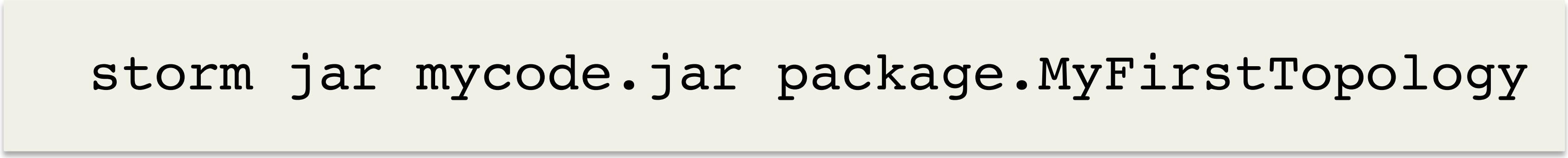
- How to implement a join with that grouping primitives?
- Have to ensure right data is ending up at nodes for the join (remember MapReduce?)

```
builder.setBolt("join", new MyJoiner(), parallelism)
  .fieldsGrouping("1", new Fields("joinfield1", "joinfield2"))
  .fieldsGrouping("2", new Fields("joinfield1", "joinfield2"))
  .fieldsGrouping("3", new Fields("joinfield1", "joinfield2"));
```

this as other pattern examples:
https://github.com/nathanmarz/storm/wiki/Common-patterns

Start a Storm Topology

- Submit jar file of your code (with dependencies) to cluster (nimbus)

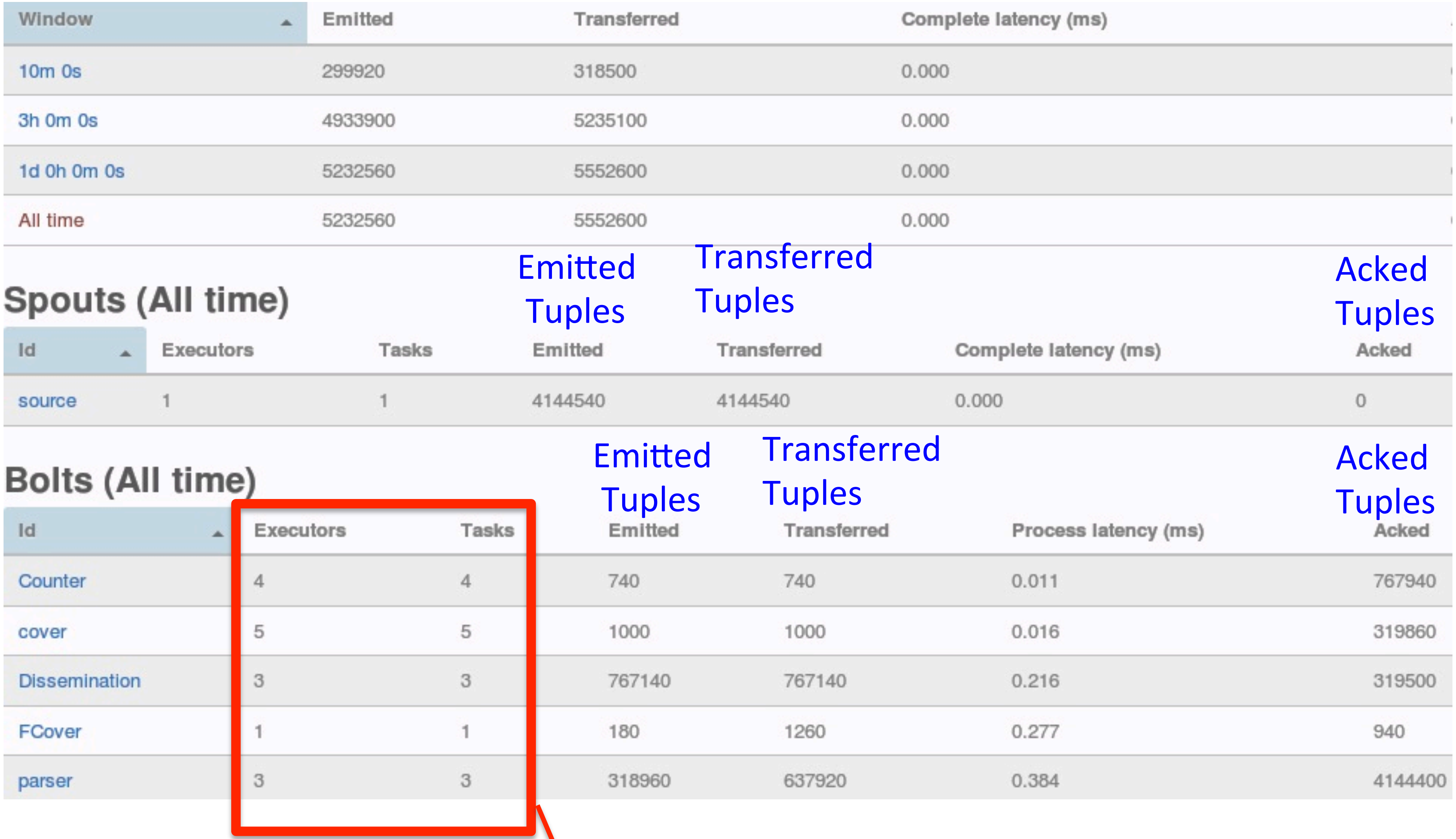


```
storm jar mycode.jar package.MyFirstTopology
```

- Looks familiar? Have seen similar usage before in Hadoop

- But: Topology will run (generally) forever, once deployed
- Can kill, monitor it

Storm UI: Screenshot



Window	Emitted	Transferred	Complete latency (ms)
10m 0s	299920	318500	0.000
3h 0m 0s	4933900	5235100	0.000
1d 0h 0m 0s	5232560	5552600	0.000
All time	5232560	5552600	0.000

Spouts (All time)

Emitted Tuples

Transferred Tuples

Acked Tuples

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked
source	1	1	4144540	4144540	0.000	0

Bolts (All time)

Emitted Tuples

Transferred Tuples

Acked Tuples

Id	Executors	Tasks	Emitted	Transferred	Process latency (ms)	Acked
Counter	4	4	740	740	0.011	767940
cover	5	5	1000	1000	0.016	319860
Dissemination	3	3	767140	767140	0.216	319500
FCover	1	1	180	1260	0.277	940
parser	3	3	318960	637920	0.384	4144400

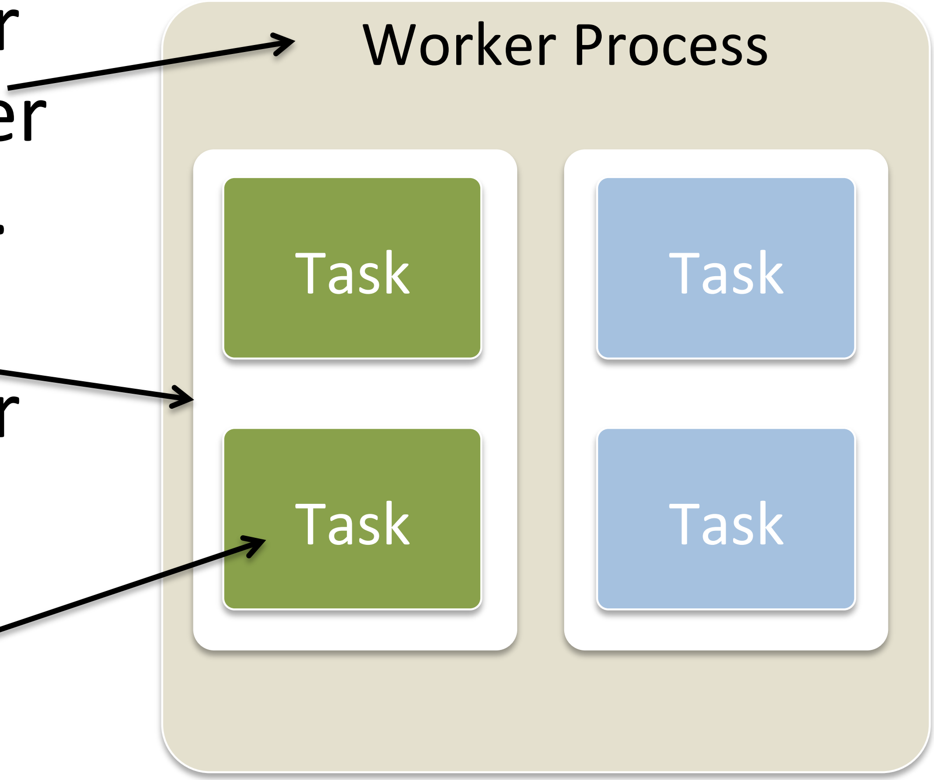
Overview of parallel execution of operators

start with command “storm ui”

default port 8080

Workers, Executors, and Tasks

- One or more worker per machine. Worker specific to topology.
- One or more executor per worker
- It runs one or more tasks of the same component (bolt/spout)



Parallelization

```
Config conf = new Config();
conf.setNumWorkers(2);
```

- use two worker processes

```
topologyBuilder.setBolt(”MyBolt",
    new MyBolt(), 2)
      .setNumTasks(4)
      .shuffleGrouping(“MySpout");
```

- Run MyBolt with 2 initial executors and 4 tasks.
- Will run two executors with 2 tasks each.
- Default is 1 task per executor.

Rebalancing a Running Topology

- Reconfigure the topology "mytopology" to use 5 worker processes
- The spout ”MySpout" to use 3 executors and
- # the bolt ”MyBolt" to use 10 executors.

```
command line: storm rebalance mytopology -n 5
         -e MySpout=3
         -e MyBolt=10
```

http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/

Fault Tolerance

- When worker dies it is automatically restarted
- If node dies, workers will be started on different machine

- Nimbus and Supervisor (daemons) are stateless (state is in Zookeper or on disk), need just be restarted
- Contrast to Hadoop: running jobs are not lost

Guaranteed Message Processing

- Consuming operators (bolts) should acknowledge the correct processing of arriving tuples; using the ack method.
- Producing operators have, thus, chance to see if tuple was properly processed
- After timeout, tuple can be resend.
- Or instead of timeout, use fail method directly

Anchored vs. Un-Anchored

- When emitting a tuple, it can be connected to its "parent" tuple; *through parameter to emit method.*

```
_collector.emit(tuple, new Values(word));
```

- Called anchoring in Storm.
- Doing so, generally, ancestor tuples of a failed tuple can be replayed.

- Tuple can have multiple anchors

- Use of special "acker task" that keeps track

Trident

- Guess what? There is a high-level abstraction on top of Storm.

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts =
     topology.newStream("spout1", spout)
       .each(new Fields("sentence"),
                  new Split(), new Fields("word"))
       .groupBy(new Fields("word"))
       .persistentAggregate(new MemoryMapState.Factory(),
                                new Count(), new Fields("count"))
       .parallelismHint(6);
```

https://github.com/nathanmarz/storm/wiki/Trident-tutorial

(MOBILE) SENSOR NETWORKS / APPLICATIONS OF DATA MANGEMENT

Recall: Sensor Networks as Data Streams Origin

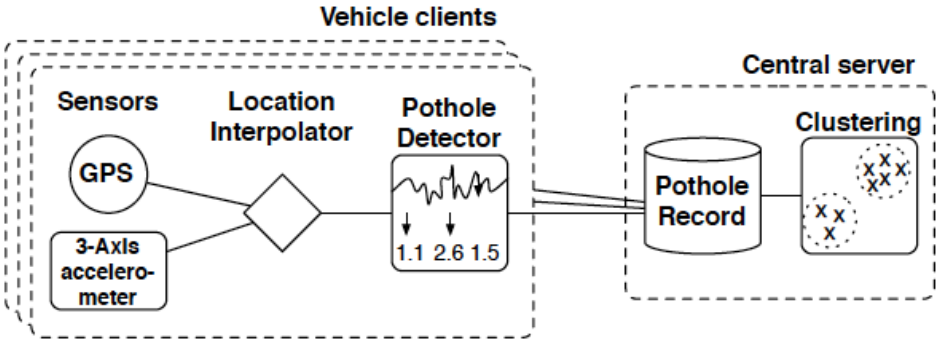
- E.g., in Environmental Monitoring

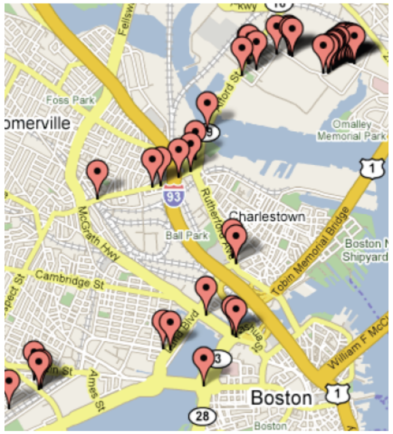
StationStream(humidity, solarRadiation, windSpeed, snowHeight)

- Various application scenarios:
  - avalanche risk level computation
  - insights for agriculture
  - air pollution (urban) monitoring

Sample Applications: Pothole Patrol

- The Pothole Patrol
- Detecting and reporting the surface conditions of roads; using sensors in vehicles
- Using 3-axis accelerometer+GPS + learning



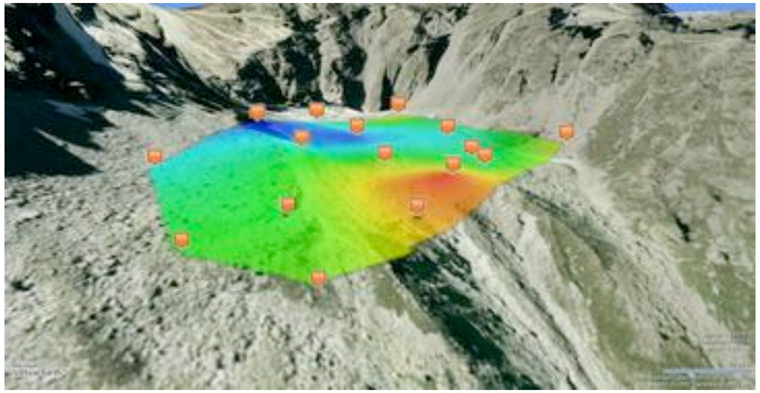


Eriksson et al. The Pothole Patrol: Using a Mobile Sensor Network for Road Surface Monitoring. MobiSys 2008.

Sample Application: Swiss Experiment

http://www.swiss-experiment.ch

- Environmental monitoring
- Sensor data management and meta data sharing.
- Across many different types of measurement:
  (hydrology, alpine monitoring, atmospheric phenomena, earthquakes, …)



- Also higher level applications like putting sensors and interpretations on maps, computing statistics over streams.

Sensors

- Sensors generate data (that we can process as previously explained, by DSMSs)

- Arbitrary application cases
- Tailored sensing hardware

plugged at station or board

with transmission capabilities (WLAN, GPRS, …)

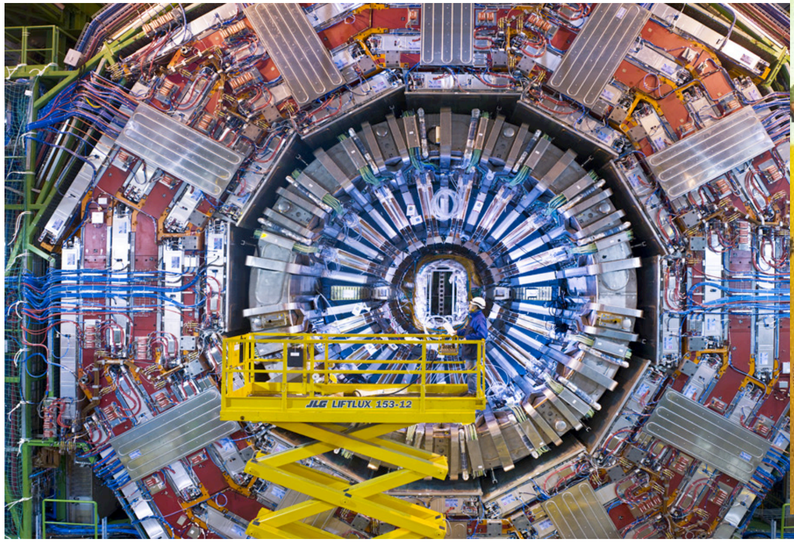
Sensors

- Mobile vs. static
- Large vs. tiny (smart dust!)
- bytes/hours vs. > GB/minute

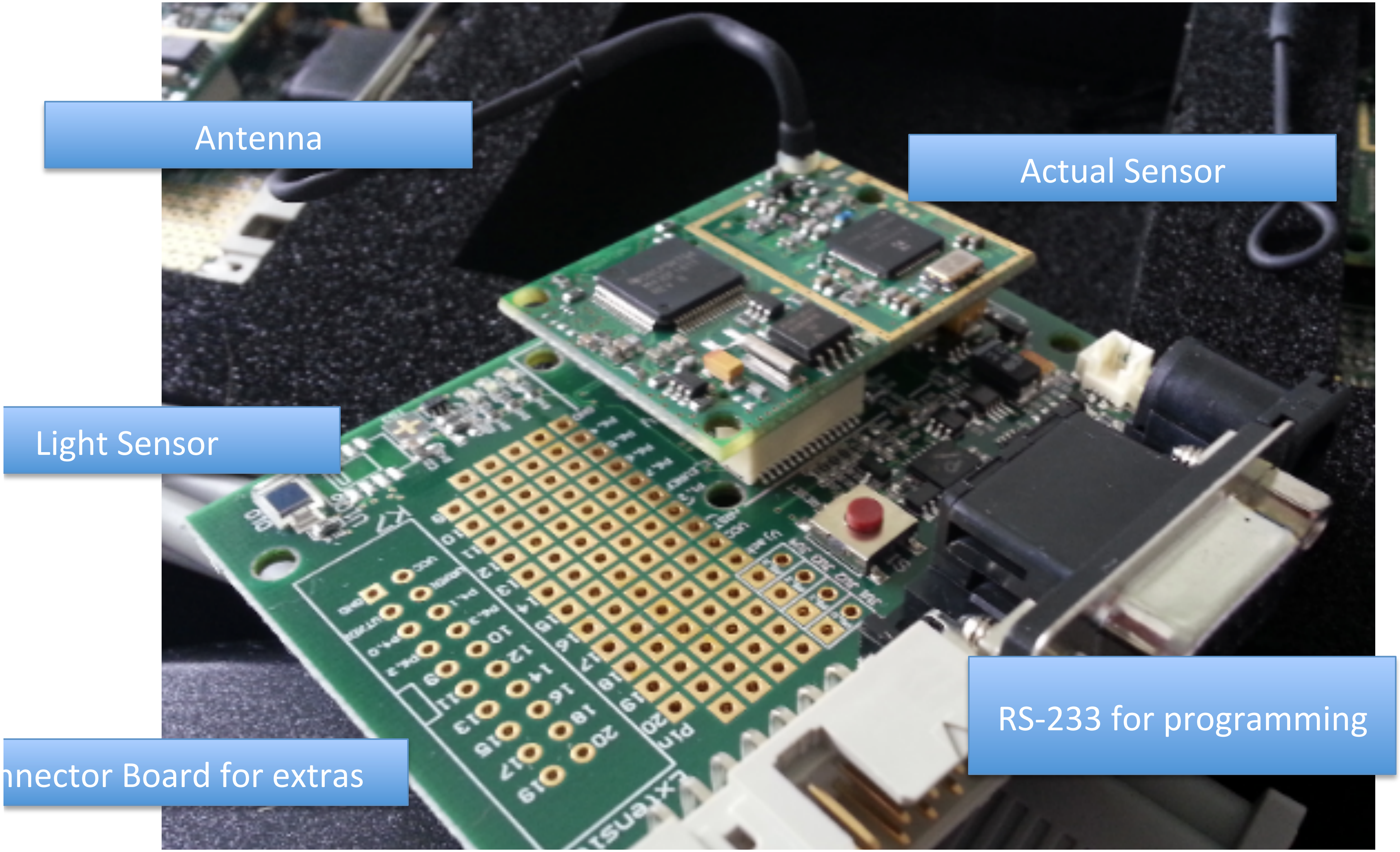
Tiny sensor at
U Michigan

ambient temp.
sensor of a car

Sensors at LHC@CERN



Example Sensor (Tinynode) on Top of Extension Board

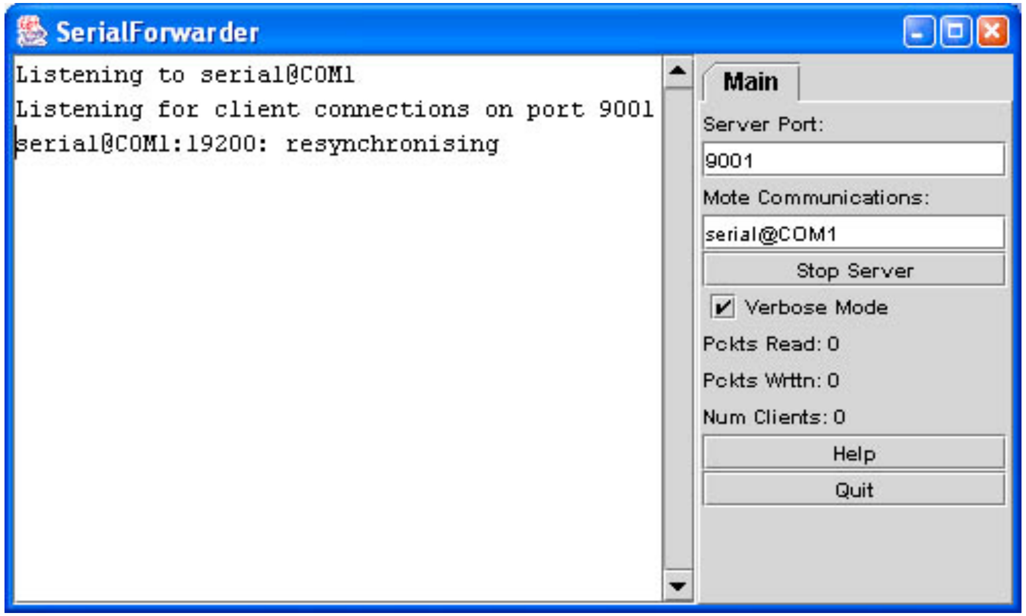


http://www.tinynode.com/

TinyOS and nesC

- TinyOS is an operating system designed to target limited-resource sensor network nodes
- nesC is a C dialect
- Program sensors (motes) to build network and measure what we are interested in

```
struct OscopeMsg
{
   uint16_t sourceMoteID;
   uint16_t lastSampleNumber;
   uint16_t channel;
   uint16_t data[BUFFER_SIZE];
};
```



```
Listening to serial@COM1
Listening for client connections on port 9001
serial@COM1:19200: resynchronising
```

Server Port: 9001

Mote Communications: serial@COM1

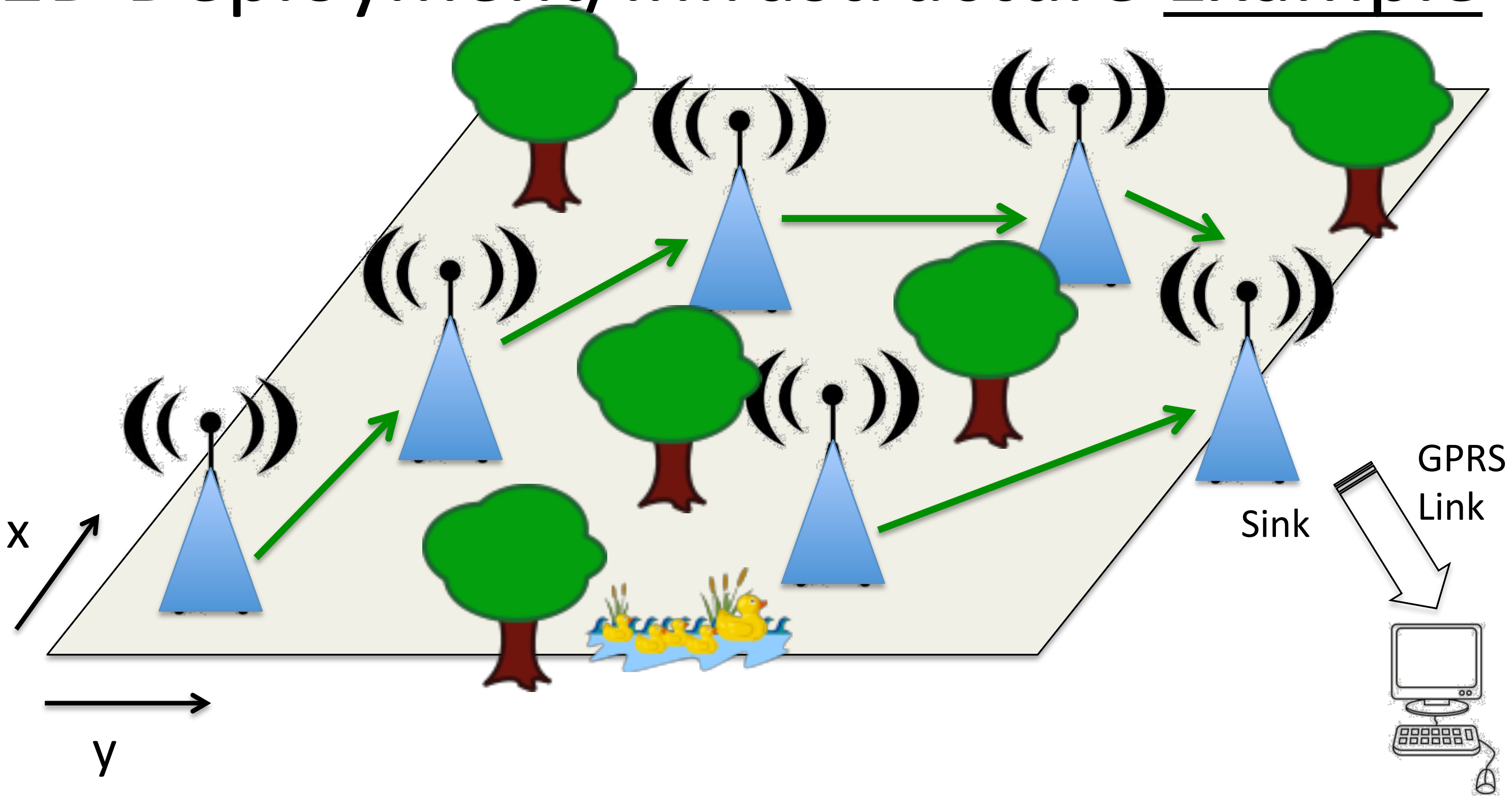
☑ Verbose Mode

Pckts Read: 0

Pckts Wrttn: 0

Num Clients: 0

2D Deployment/Infrastructure Example



- Multi-hop communication toward one sink node.
- Or direct communication to consumer
- Or Sensors might move, creating ad-hoc networks
- …

Sensor node (aka. Mote) Characteristics

- Shockfish TinyNode (a Swiss Company)
  - Texas Instruments MSP430: 16bit microcontroller, running at 8 MHz
  - Semtech XE1205 radio transceiver: max rate of 76 Kbps
  - 10KB RAM
  - 512KB flash memory

- Apparently good deal between power consumption and communication range

Challenges/Research

- Sensors usually have limited battery capabilities (although complemented with)
- as well as processing power
- and reach of radio signal for communication

SensorScope

- As one specific example of the utilization and realization of wireless sensor networks we look at SensorScope.
  - Project at EPFL (Lausanne, Switzerland), two groups (communication systems lab, and environmental scientists)
  - Environmental monitoring sensor stations; base station with several application specific sensors
  - Aims at low cost stations; easy to setup, “lighweight”

François Ingelrest, Guillermo Barrenetxea, Gunnar Schaefer, Martin Vetterli, Olivier Couach, Marc Parlange: SensorScope: Application-specific sensor network for environmental monitoring. TOSN 6(2) (2010)

Measured Quantities Example

SensorScope station

Measure	**Sensor**	**Range**	**Precision**
Air humidity	Sensirion SHT75	0−100%	±2%
Air temperature	Sensirion SHT75	-20−60°C	± 0.3°C
Precipitation	Davis Rain Collector	0−∞ mm	± 1mm
Soil moisture	Decagon EC-5	0−100%	± 0.1%
Solar radiation	Davis Solar Radiation	0−1800W/m^2	± 90 W/m^2
Surface temperature	Zytemp TN901	-33−220°C	± 0.6°C
Water content	Irometer Watermark	-200−0kPa	unknown
Wind direction	Davis Anemometer	0−360°	± 7°
Wind speed	Davis Anemometer	1.5−79m/s	± 1.5 m/s

source: SensorScope paper

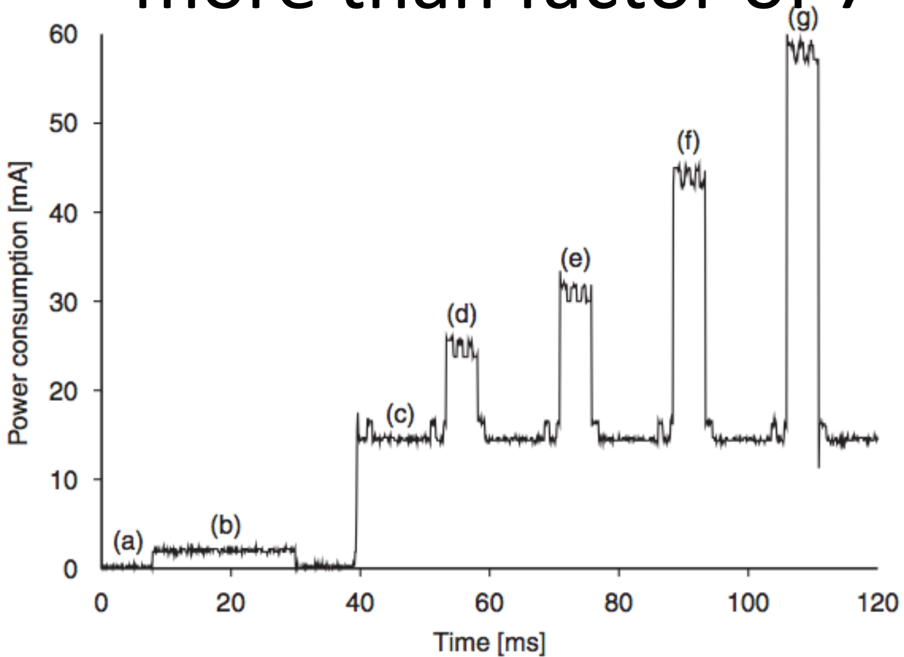
Communication

- Each node keeps table of neighbors.
- Neighbors are nodes the node can (literally) hear, by observing radio signals
- Cost of routing to sink is updated if new node is discovered

- Estimating link quality between nodes due to randomness of radio channel; by observing lost messages or signal strength

Power Management

- Radio signal is big energy consumer
- Just switching is on increases power cons. by more than factor of 7



Label	Activity	Value
(a)	None	0 mA
(b)	CPU on	2 mA
(c)	Radio on	15 mA
(d)	Transmission (0 dBm)	25 mA
(e)	Transmission (5 dBm)	31 mA
(f)	Transmission (10 dBm)	44 mA
(g)	Transmission (15 dBm)	58 mA

- Results are of course specific to actual hardware, but exemplary for behavior

image source: Ingelrest et al. SensorScope. TOSN 6(2) (2010)

Power Management (Cont'd)

- Nodes have two-state communication cycles:
  - active state (i.e., radio is on)
  - idle state (radio is off)
- Idle state should be as long as possible but still allow communication between nodes. How?
  - Low-power listening: announce packet by sending specific bit pattern (with length larger than idle state). Nodes see it and wait for packet.
  - Duty-cycling: All nodes switch radio on at same time (synchronously). Used in SensorScope. Nodes have sync'ed time anyway (so it's "easy").

Sensor Management / Middleware

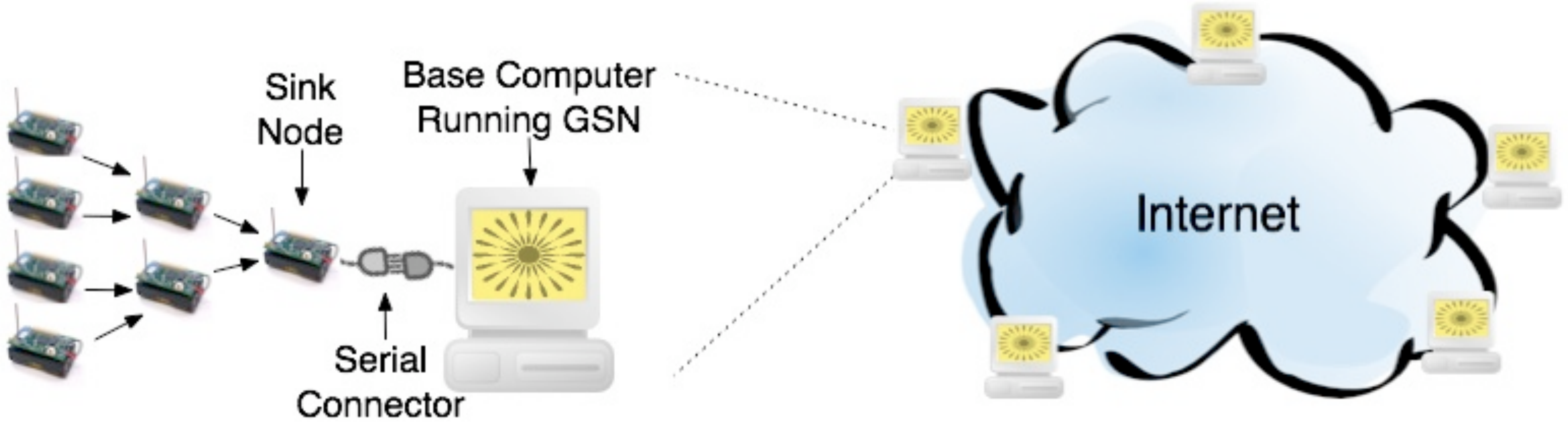
- Different sensors come with different packet formats for transmissions, also different connectors/interfaces to program against

- Abstraction needed to unify/enable usage

- In principle: make tuples/relations out data obtained through various interfaces
- Offer metadata and computation means, sharing, access control, …

Global Sensor Networks (GSN): High Level View

- Open-source sensor middleware
- Developed at LSIR lab at EPFL
- Comes with wrappers for various sensor
- And higher level operations, e.g., data cleaning, visualization
- Runs on local instance, but can connect to others



Karl Aberer, Manfred Hauswirth, Ali Salehi: Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. MDM 2007: 198-205

GSN: Virtual Sensors

- Abstraction of physical sensors or local operators (also remote)
- Specified through an XML document (mix of SQL and specification of Java Classes that act as wrappers)
- Virtual sensors can be composed of other virtual nodes; resembling operators in the operator DAG (seen before)
- Use of standard SQL to process queries over wrapper data or other virtual sensors

Receiving Data in Virtual Sensors

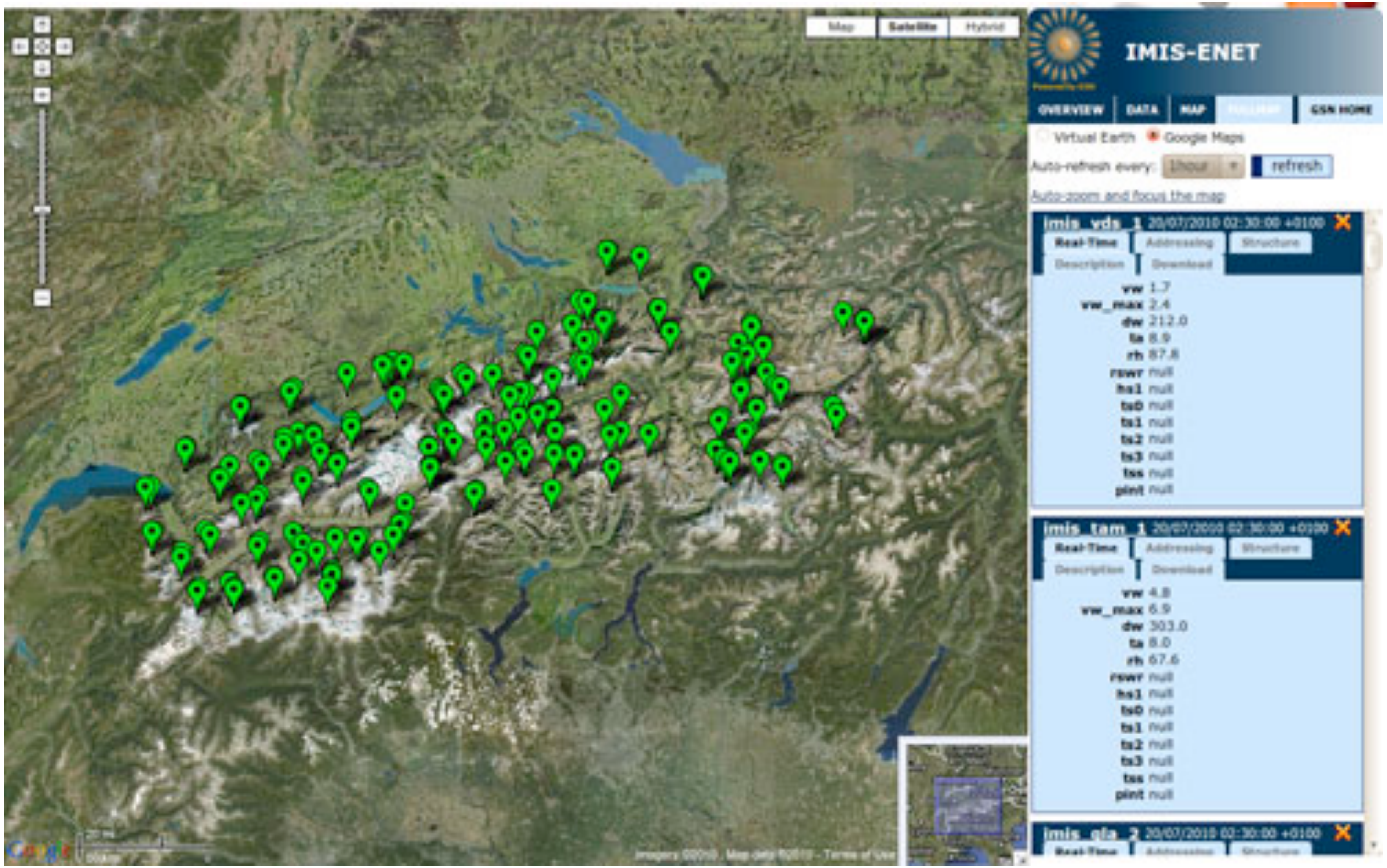
- Virtual sensor receives tuples on

```
public void dataAvailable(
     String inputStreamName,
      StreamElement streamElement ) { … };
```

- Like `execute(Tuple tuple)` in Storm

- If output is produced: Virtual sensor will notify the responsible "manager" by adding itself to the list of the virtual sensors which have produced data.

GSN UI (Map)



○ Virtual Earth ● Google Maps

Auto-refresh every: 1hour

imis_vds_1 20/07/2010 02:30:00 +0100

vw	1.7
vw_max	2.4
dw	212.0
ta	8.9
rh	87.8
rswr	null
hs1	null
ts0	null
ts1	null
ts2	null
ts3	null
tss	null
pint	null

imis_tam_1 20/07/2010 02:30:00 +0100

vw	4.8
vw_max	6.9
dw	303.0
ta	8.0
rh	67.6
rswr	null
hs1	null
ts0	null
ts1	null
ts2	null
ts3	null
tss	null
pint	null

imis_gla_2 20/07/2010 02:30:00 +0100

http://sourceforge.net/apps/trac/gsn/

Side Remark: Sensor Metadata

- Sensor data has important additional (often static) aspects besides pure measurement values
  - sensor manufacturer, measured quantities, units
  - frequency of measurements, averaging applied, sampling?
  - sensor serial number
  - geographic location
  - quality information
- Why? Essential to make good use of data.
- Overall, want full lineage (aka. provenance): where does data come from, what happened to it (transformations, errors, etc.)

Related Areas (Subset)

- In-network data processing
- Communication efficient data gathering (optimizing also for battery lifetime, not necessarily query response time)
- @KL: distributed computer systems lab
- From “our” perspective the work mainly starts when data is at hand:
  - But might be noisy (uncertain) or incomplete
  - Data mining: finding patterns, trends, predicting behavior, such as in road networks, people movement at festivals, etc.
  - Computation of complex models, like spatial interpolations, inference

Literature

- Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, Stanley B. Zdonik: The Design of the Borealis Stream Processing Engine. CIDR 2005: 277-289
- http://storm-project.net/ and documentations/tutorials within
- http://zookeeper.apache.org/ and documentations/tutorials within
- Karl Aberer, Manfred Hauswirth, Ali Salehi: Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. MDM 2007: 198-205
- Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, Hari Balakrishnan: The pothole patrol: using a mobile sensor network for road surface monitoring. MobiSys 2008: 29-39
- Karl Aberer, Gustavo Alonso, Donald Kossmann: Data management for a smart earth: the Swiss NCCR-MICS initiative. SIGMOD Record 35(4): 40-45 (2006)