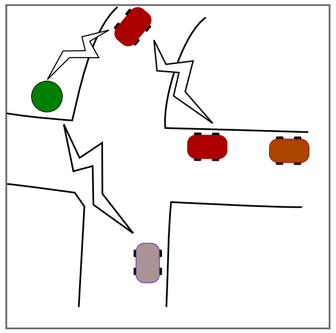
Embedded Motion Control

Best Practices in System Design for Robot Control

Component
(Barrier, lights)



System
(Traffic lights)

System of systems
(cars coordinate themselves)

Nico Huebel & Herman Bruyninckx

Mechanical Engineering
KU Leuven, Belgium

Objectives of this lecture

To understand the software design for System of systems: **information architecture** and **software architecture**.

Discussion about “control”:

- state of cars & traffic?
- motion control in single car?
- motion coordination between cars?
- what traffic control is needed?
- what other “control” is needed?

Remember: program = functions + data structures + control flow

Objectives of this lecture (2)

Important new focus that should be clear from first two slides:

- “control” of Systems of systems requires **dialogues** between the sub-systems, and **not just data and event** communication.
- such dialogues can only be held if all participating systems **share a common language about how “their domain works”**.
- such a language represents, formally, the **Primitives**, **Relationships**, **Constraints**, and **Tolerances** of the domain. For example, in traffic:
  - *Primitives*: cars, lanes, stop line, roundabout, pre-sorting lanes, direction indicator, etc.
  - *Relationships* between Primitives: driving in a lane, changing lanes, priority in lanes, priority between cars of different types (e.g., emergency vehicles), driving in queue, traffic jam, use of direction indicator in lane change, protocol aboutn how to enter a roundabout, etc.
  - *Constraints* on the properties of Primitives and/or Relationships: minimal distance between cars, speed limits on roads of various types, etc.
  - *Tolerances* on Constraints: safe driving vs max-throughput driving, fine tolerance on speed limit violations, etc.

Three modes of *programming*

- **Programming in the Small** — “Component”
  - one single programmer has *full* control of data, functions, and control flow.
  - whole programme in one programming language.
- **Programming in the Large** — “System”
  - multiple programmers/programmes/languages, but *developed in team*.
  - programmers take care of *application resources*
    OS (asynchronously!) *controls* sharing of the “CPU”, “BUS”, “RAM” resources.
- **Programming in the Extra-Large** — “System of systems”
  - multiple companies develop multiple Systems, *independently*.
  - each System *cannot control* its “peers”, to provide data and to execute functions, asynchronously over the *Internet*.
  - each System has (a lot of) code *to be robust* against the others *not* doing what is expected.

Why is step from “Component” to “System” difficult?

1. Asynchronicity:
state can be changed by other Components in the System, **beyond the control** of one Component's “control flow”.

> ***Example****: position + velocity state of moving car is estimated with data from several sensors interfaced through (local) network, and provided by devices with their own computer units.*

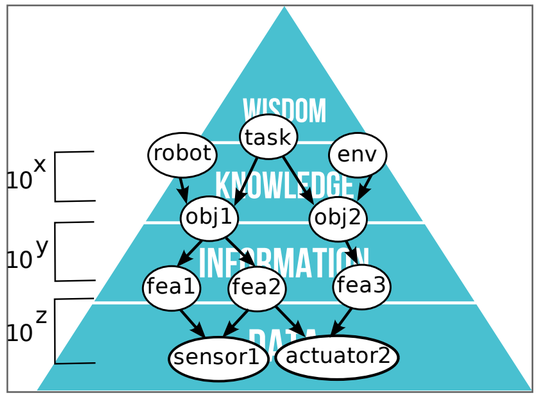
⇒ making **decisions** based on *possibly inconsistent* state becomes **undeterministic**.
⇒ **software architecture** needed for *determinism*.

2. Semantics:
interpretation of data structures can differ between programmers of different Components.

> ***Example****: position + velocity state of moving car represented by three floats for position, and three floats for velocity, but Component A is using meters and radians, while Component B uses yards and degrees.*

⇒ *decisions* could be made on **wrong interpretation** of *state*.
⇒ **information architecture** needed for *unambiguity* of interpretation.

Information architecture for unambiguous Semantics



Examples of Knowledge:

- traffic rules
- traffic queues and jams
- load escaping from robot gripper falls down
- recognize airplane stall
- …

Examples of Information:

- traffic rules to be taken into account *now*.
- cars in queues & cars in jams.
- impedance properties of object in gripper.
- state of stall-related parameters.

Examples of Data:

- position logs of cars.
- force & position logs of gripper.
- measurement data from stall-related sensors.

Formal models (*"Domain-Specific Languages"*, DSLs) must be introduced to connect data to information and to knowledge.

(The triangle in the figure above is called the ***Knowledge pyramid***.)

Information architecture for Traffic System of systems

How **to represent** the various **state** variables:

- of Components.
- of System.
- of System of systems.
- of the *interactions* between all of the above.

How **to represent** the various **actions**:

- what does one car have to know about the state of the others before it can take (safe and predictable) decisions about entering a traffic point?
- *what* does it have *to know* about the state of the traffic point?
  (which is where the *cooperative interaction* between cars takes place!)
- *what* does it have *to do* with all the state values?
- *when* does it have to do that?

Information architecture for Traffic System of systems (2)

Pragmatic way to create DSLs

Take a couple of people and “play” the game of being cars in traffic yourselves:

- identify what you say to each other to make traffic work, and work safely;
- identify what models about traffic that you have in your mind, and that you expect to be present in the other people's head too.

You probably will find out that the **non-nominal** cases require a lot more attention and “language development efforts” than normal traffic.

Information architecture for unambiguous Semantics

The following slides introduce **two models** of how **to compose** a (mechatronics, or other Cyber-Physical) *System of systems* from *Components* and *Sub-Systems*.

The first model, the **“Composition Pattern”**, identifies which **roles and responsibilities** (the “5Cs”) appear in all systems, and how they are playing together.

The second model, the **“Four Levels”**, helps you to identify which **resources and capabilities** appear in your systems, and how they can be put together via the Composition Pattern.

Modelling the domain-specific behaviour in your application systematically, via **hybrid constrained optimization problem** formulations, supports the **integration** between different Levels, as well as within one single Level: the components that must work together provide each other the *objective functions*, *constraints*, and *tolerances* that they would like the other components to take into account.

Hence, solving *hybrid constrained optimization problems* is a major **Computation**al activity in the system, together with the **monitoring** of the controllers that execute the outcomes of the optimization, and the **Coordination** and **Configuration** that can be triggered by events generated in the monitoring.

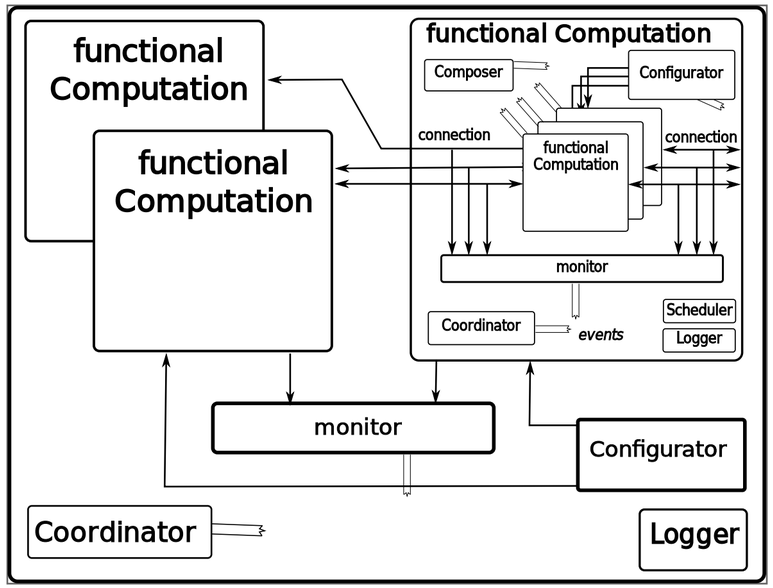
Information architecture for unambiguous Semantics (2)

Coordinating the activities of all interacting components requires a **common language** to hold **dialogues** about *setting up* a cooperation, about *monitoring the progress* of that cooperation, and the *quality of its execution*.

The requests that components communicate to each other should always be considered as just that: *requests*. And not as *commands* that are expected to be executed in a guaranteed way.

An important part of the design of such a language is to formalize the **set of events** via which components inform each other about the progress of their agree-upon cooperation.

Composition Pattern for “5C” functionalities



Computations — *continuous state*:

- dynamics of machines
- change in properties of objects/environment
- *Logger*, *Monitor*, *Scheduler*: recurring types of Computations

Configuration — *changing Computations state*: values of parameters in Computations

Coordination — *discrete state*: decision to change which Computations and Configurations are activated.

Communication — *connecting states*

Composition:

- design of all of the above components, as one System
- same design *pattern* reused *fractally*

Composition Pattern for “5C” functionalities (2)

The **fractal** property of the Composition Pattern means that:

- it repeats itself *“inwards”*: a **higher level of detail** model for any primitive in an existing model will be a “full” Composition Pattern in itself.

  For example, a motion control system can consider the torques to be sent to actuators as its most primitive data blocks, but in reality they are always just inputs to control loops (e.g., current control) around the actuators.

- it repeats itself *“outwards”*: the design of the **integration** of an existing system into a larger system of systems will also be done best via the Composition Pattern.

  For example, a robotic workcell can be “just” one of the many devices that have to cooperate in a production line, and that line can be “just” one of the many that have to cooperate in a manufacturing plant.

In all these cases, the levels are coupled by the exchange of (actual/desired) “quality of service” data, functional data, events, and objective functions or constraints, following the methodology of composition advocated in the Patterns.

Composition Pattern for “5C” functionalities (3)

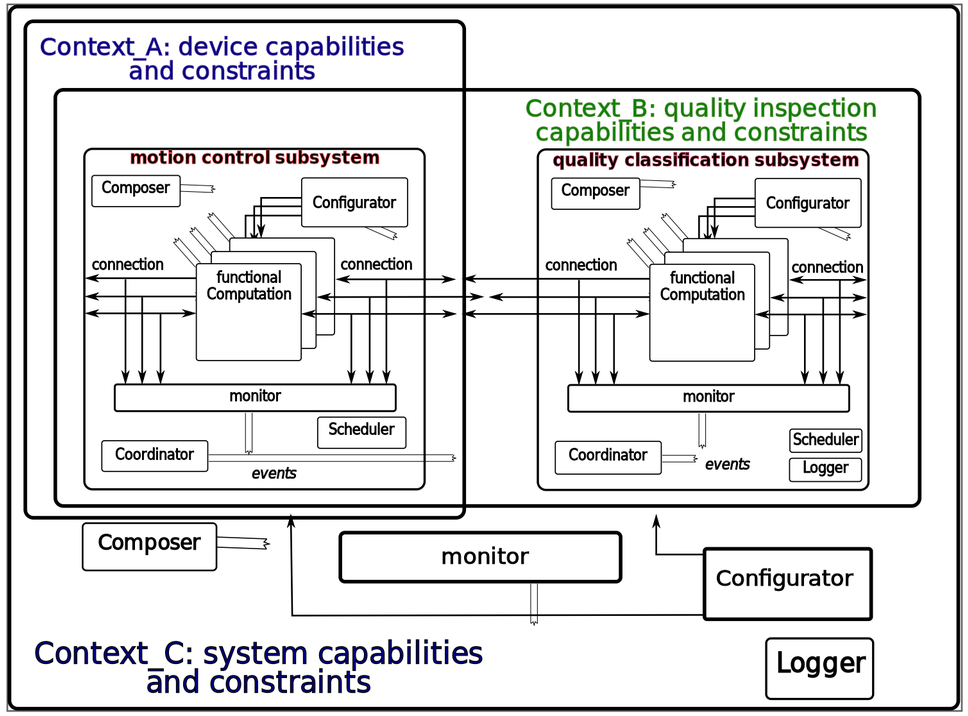
In both mentioned “design compositions”, there are always(?) several opportunities for optimization in the information model because:

- the Coordinators of composed sub-Systems might be fused into the Coordinator of the composing System.
- the same holds for the Configurators, and for the “support” components (Monitors, Schedulers, Loggers).
- similarly with Communication, but this will often require extra **decision making about filtering messages**, depending on the current status of the System, the **Quality of Service** that its **Capabilities** can offer to its peers, or that of the **resources** that it relies on itself.

Such “optimizations” quickly become highly domain-dependent.

“Domain-dependent” implies *“knowledge contexts”*, and the following slide shows how to integrate such contexts into the Composition Pattern.

Composition Pattern with knowledge contexts



Knowledge:

- values of parameters in the Information models in various Components **depend** on the same particular **context**.
- contexts can **overlap**.

Examples:

- motion control of driving must be adapted to normal traffic, queues, jams, one-lane to four-lane driving,…
- tunings of **Engine Control Unit** and **Electronic Stability Control** are adapted to driver style: eco, comfort, sportive,…

Practice: the contextual knowledge about parameter values typically ends up in “magic numbers” in the various Configurators within the scope of the context.

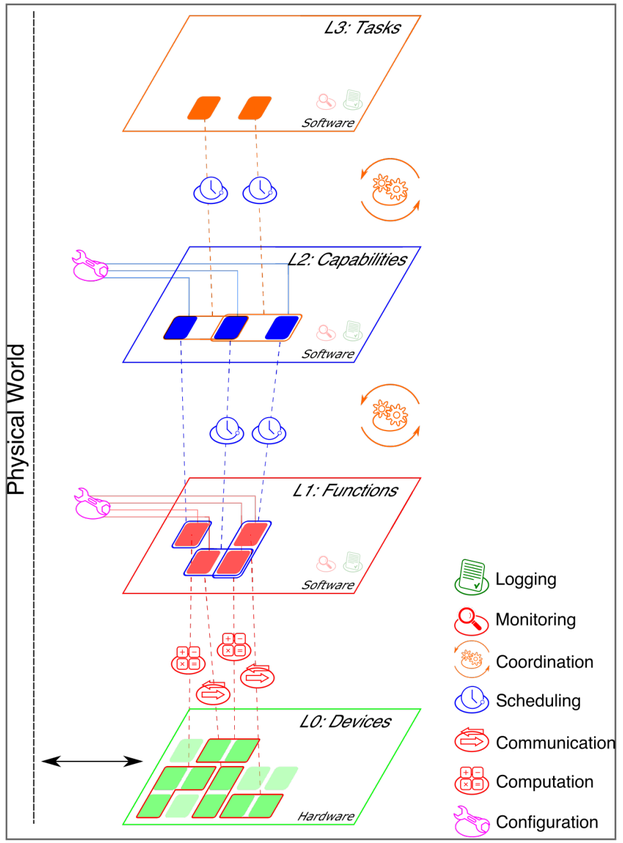
Four-Levels Pattern of “5C” Information architecture

Levels:

- Tasks
- **Capabilities**
- Functionalities
- Devices/Resources

Information dependencies (“context”):

- Scheduling, Coordination and Configuration typically only come into play in Capabilities, because that is where it begins to make sense to execute different functions in particular order and with particular settings.
- similarly, *Tasks* depend on similar particular settings in the *Capabilities* they use.
- the start of a *Task* can depend on the outcome of other *Tasks*.
- *Capabilities* have **software architectures** that depend on the *hardware architecture* of *Devices*.
  None of the other levels requires software architectures.



From Information architecture to Software architecture

The models of the system designs introduced up to now are just that: *models*.

What is missing is “something” to turn the information architecture into something that can be *executed*.

That “something” is called a **software architecture**, which uses the primitives offered by programming languages and operating systems.

It integrates the information architecture with the hardware available for (i) **communication** (referred to in a generic way as *“BUS”*), (ii) **computation** (referred to in a generic way as *“CPU”*), and (iii) **memory** (referred to in a generic way as *“RAM”*).

It makes a lot of sense to look at this integration as just another example of the composition of *“Capabilities”* with *“Resources”*, hence, the same model terminology and patterns can (should!) be reused.

Software architectures are not a focus of this course; information architectures are, and especially how to make them with the patterns introduced before.

Software architecture: sharing “state”

Building blocks:

- programming in the Small: **threads**, that is, serial control flows whose functions are executed on the *same CPU*, and have access to data structures in the *same RAM*.
- programming in the Large: **processes**, that is, sets of threads that are *shielded* from each other's RAM, and hence need at least one BUS to cooperate; it is *possible* that different processes are executed by the same CPU.
- programming in the Extra Large: **services**, that is, sets of processes that interact with each other while being *somewhere* at an unspecified location on the Net, hence not sharing any RAM or CPU.

Only in a *thread*, **synchronous programming** is possible, which means that:
there is *one single* control flow, and *each function* runs to completion.
Hence, the order of access to data is clear from the programme code.

Systems always require **asynchronous programming**:

- *multiple* control flows (“threads” → “services”);
- execution of functions can be *pre-empted*;
- on single machine, or distributed over a network.

→ the order of access to data is *not clear* from the programme code.

Inter-Process Communication: how & why?

How?

- **message passing interface:**

```
send(message,channel);
```

```
read(message,channel);
```

- **streams interface**: like message passing, but caller is ready for the case there is no data yet, no data anymore, or multiple answers at the same time.

Why?

- *data streams* between identified data processors;
- *coordination events* between all processes;
- while making sure that:
  - no data is lost (unless application wants it!);
  - no data is corrupted;
  - access to data is efficient.

  → all of these reasons have a performance that (should) depend on the application!

Bad practices in multi-processing

Avoid <u>mutexes</u> or locks:

- these are OS-facing programming primitives, that can be misused in way too many ways…
- use a library that (i) provides abstraction of access to shared resources, and (ii) with configurable application-facing performance.

Avoid <u>priorities</u> to determine “which process goes first”
→ use events-with-meaning!

Avoid to share state explicitly,
→ use information *events* about other processes’ *state changes* instead!

Best practices in multi-processing

One event queue in each process, hence:

- **one Coordinator**:
  - takes decisions on *what* to do:
  - based on interpretation of *incoming* events:
  - decisions are provided in the form of *outgoing* events.
- **one Scheduler**: defines *in what* order to call functionalities.
- **one Configurator**: defines all “magic numbers”.

Multiple data queues:

- configured via events;
- available as fully accessible “streams”.

Multiple Computations:

- configured via events:
- “control”, “monitoring”, “world modelling”, “trajectory selection”,…;
- take data/event flows in, put data/event flows out.

Best practices in multi-processing (2)

Each synchronous activity has one, and only one, “event loop”

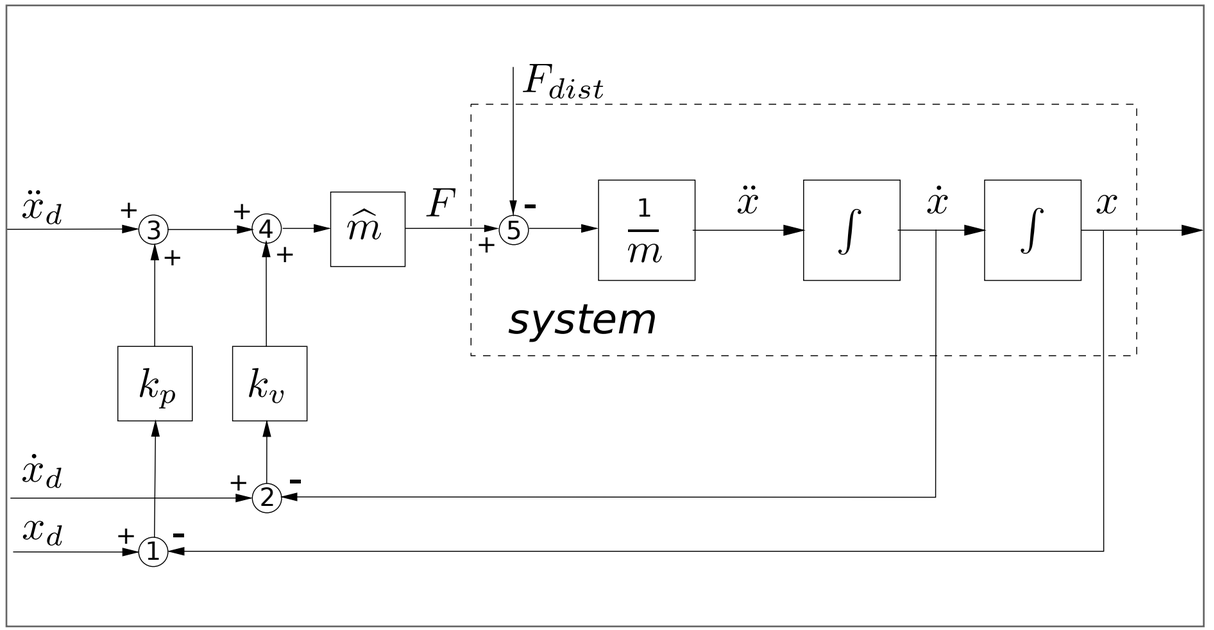
```
when triggered // = scheduling by OS
 do {          // = your own scheduling
  communicate()  // get "message" with events & data
                 // = link with all other
                 //   asynchronous activities
  coordinate()   // handle the events
  configure()    // possibly requiring reconfiguration

  schedule()     // run your own Computation functions

  coordinate()   // functions could trigger new events
  communicate()  // or data that others might
                 // have to know about
  log()          // remember what happened
 }
```

Example of event loop: one-dimensional position controller

- *state* of the system $x(t)$, i.e. position of the mass m.
- *setpoint inputs* $x_d, \dot{x}_d, \ddot{x}_d$, e.g. desired position, velocity and acceleration of the mass.
- *measurement inputs* x and $\dot{x}$, e.g. actual position and velocity of the mass.
- *feedback gains* k_p, k_v, i.e. proportional position/velocity control gains computed, for example, via pole placement (off line).
- *outputs* of control is desired acceleration, reached after summation “4”.
- feedback output is transformed, via *feedforward* multiplication by the *estimated* mass $\hat{m}$, into force F to system actuators.
- *disturbance* force F_{dist} applies after control, at summation “5”.
- *system*: assumed to be a perfect double integrator with real mass m.



Event loop for one-dimensional position controller (2)

The following *function blocks*, *data blocks* and *control flow schedule* realise the controller.

- *data blocks*: each arrow in the control diagram represents a dedicated data structure, whose meaning and structure is clear from the control diagram.
- *function blocks*: each rectangle in the control diagram represents a multiplication, and each circle represents a summation.
- the *schedule* that realises the controller's *event loop* (by triggering *function blocks*) is the following:

```
when triggered // = OS executes controller every, say, 10 milliseconds
 do {
  communicate()  // read desired position/velocity/acceleration
                 // from input data block(s)
                 // read actual position/velocity from sensors
  schedule()     // trigger function blocks, in the following order:
                 // sums 1 & 2, multiplications  k_p &  k_v,
                 // sums 3 & 4, multiplication  m̂
  communicate()  // write computed control force to actuator data block
 }
```

Event loop for one-dimensional position controller (3)

It is possible that the computation of the control loop generates *events* itself. (Or rather, that are generated in *monitor* functions that the `schedule()` function adds to some data blocks in the controller.) For example:

- when an *error* between desired and actual state parameters is too large.
- when the *trend* of the error is undesired, e.g., always positive.
- when the computed control force F is too large for the actuators.

It is possible that the computation of the control loop must react to *events* that come from the outside (and that are different from the *timer events* that most control loops rely on). For example:

- a new trajectory is started, so that the control must be reset.
- the current trajectory is interrupted, so that the controller must bring the system to a safe stop as quickly as possible.

Hence, the control loop event queue must be extended with `coordinate()` functions to react to (and/or generate) events, and `configure()` functions to realise the reconfigurations triggered by the coordination execution.

The “safe stop” functionality would require the addition of extra functions blocks. (This is preferably realised *outside* of the position control anyway.)

Event loop for one-dimensional position controller (4)

Notes

- implementations of control loops typically require no “real” communication, but just reading and writing from data in the memory of the computer. The core technology here is **memory-mapped IO**.
- more and more control systems (and even components) use **field busses**, with **Ethernet-based protocols** being the most popular ones. Such architectures require at least two asynchronously running activities: the field bus **device driver** takes care of the communication over the network, and writes/reads messages into the `data blocks` that the controllers use in the their event loops.
- **interrupts** are another very important source of events in control systems; many modern interface devices can be configured to generate events to which the operating system will react. The application can configure the operating system to schedule a specific **interrupt handler** function as soon as the interrupt arrives. (The above-mentioned communications most often work in such an interrupt-driven way.)
- the software offered in the course to do software projects is using the event loop pattern everwhere, but almost never in its full form but always with context-specific simplifications.
  Your extensions and improvements should do the same.

ZeroMQ library for asynchronous programming,

takes care of “discovery” & “communication”

Key reference: *“Multithreading with ZeroMQ”*
(http://zguide.zeromq.org/page:all#toc45)
Also contains code examples in many programming languages.

Data streams: ZeroMQ sockets provide an *abstraction* of asynchronous message streams, multiple messaging patterns, message filtering, seamless access to multiple transport protocols and more.

That abstraction is good enough for this course :-)

Discovery & group communication with *Zyre* sub-library:

- *whisper* to identified peer in a *group*.
- *shout* to all peers in a group,
- one peer can take part in several group communications.

Image references

All images in this presentation are taken from Wikimedia Commons, except the ones made by the author. Zhang Lin is the author of the “Four Levels” figure.