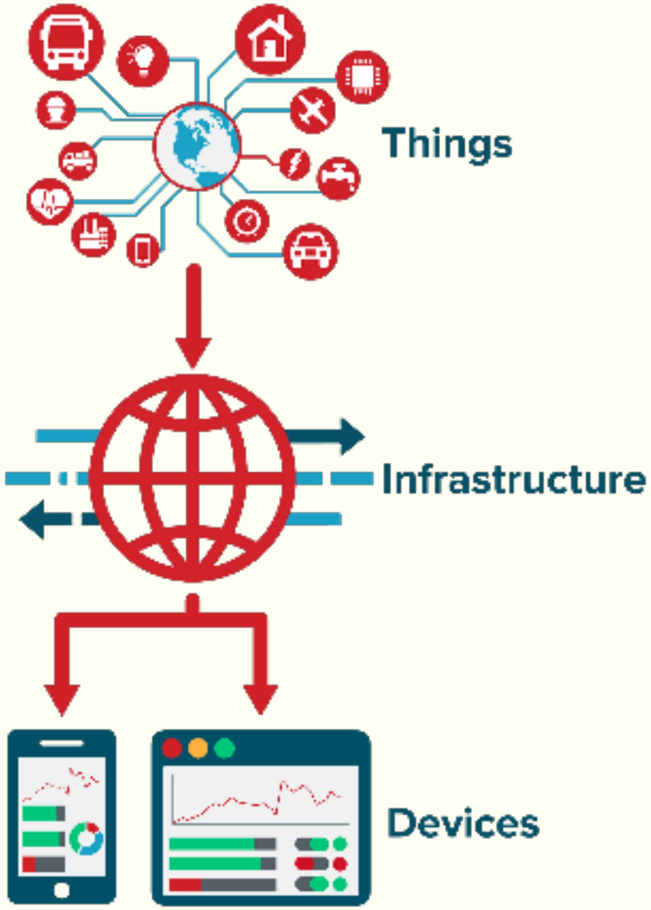
TARGETING SCALE-OUT AND SCALE-UP PLATFORMS WITH A UNIFIED PARALLEL PROGRAMMING MODEL

Nicolò Tonci
nicolo.tonci@phd.unipi.it

Agenda

- Context & State of the art
- *FastFlow*
- The distributed-memory RTS for *FastFlow*
- Launcher module
- Hands-on - Demo
- Future directions
- Q&A

Context



~41 billion devices in 2025

Big Data Analysis Trend

Need for new highly scalable applications and programming interfaces for shared-memory and distributed architectures.

Use cases:

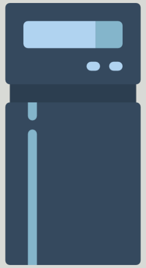
- Smart cities
- Event forecasting
- Financial Analysis

State of the art

Single machine frameworks

(Shared-memory)

- Very efficient use of local resources
- Can exploits hardware acceleration
- No support for multiple machines
- E.g., ***FastFlow***, *Intel TBB*, *OpenMP*

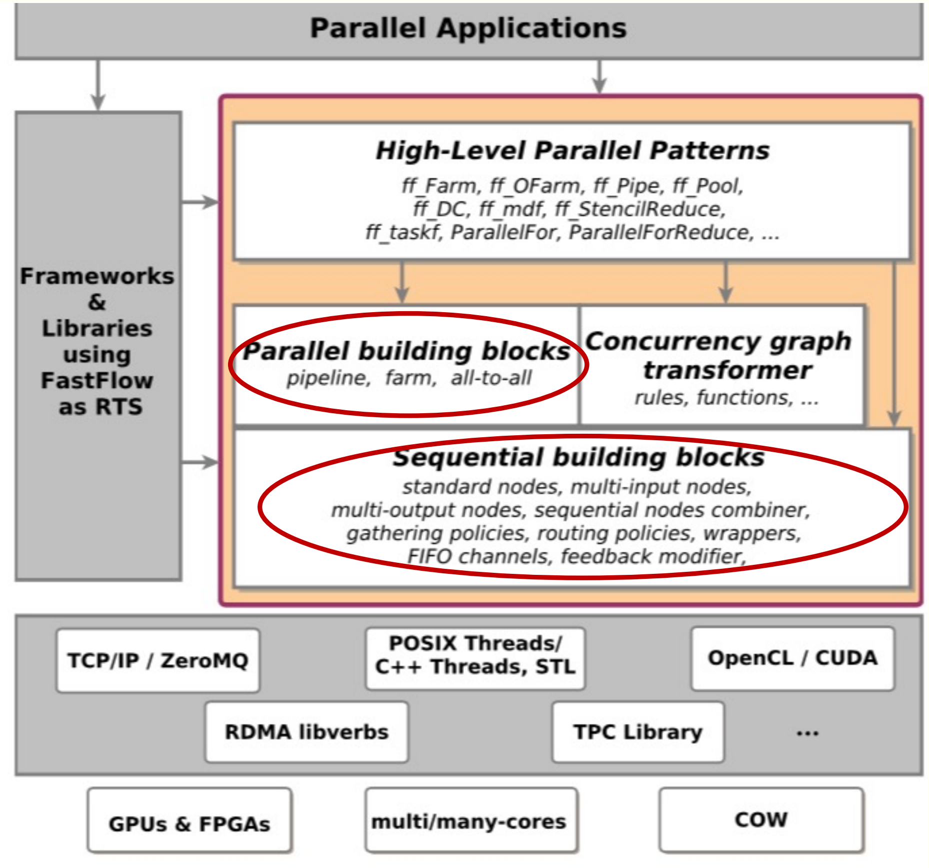


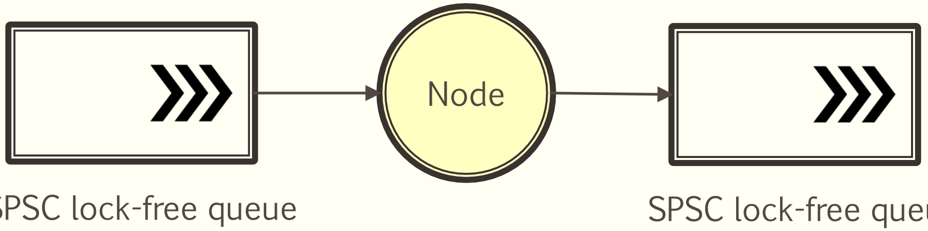
Distributed systems frameworks

- Designed to handle hundred of machines
- Very heavy frameworks
- Poor performance within the single machine
- E.g., *Apache Storm*, *Apache Spark*, *Apache Hadoop*

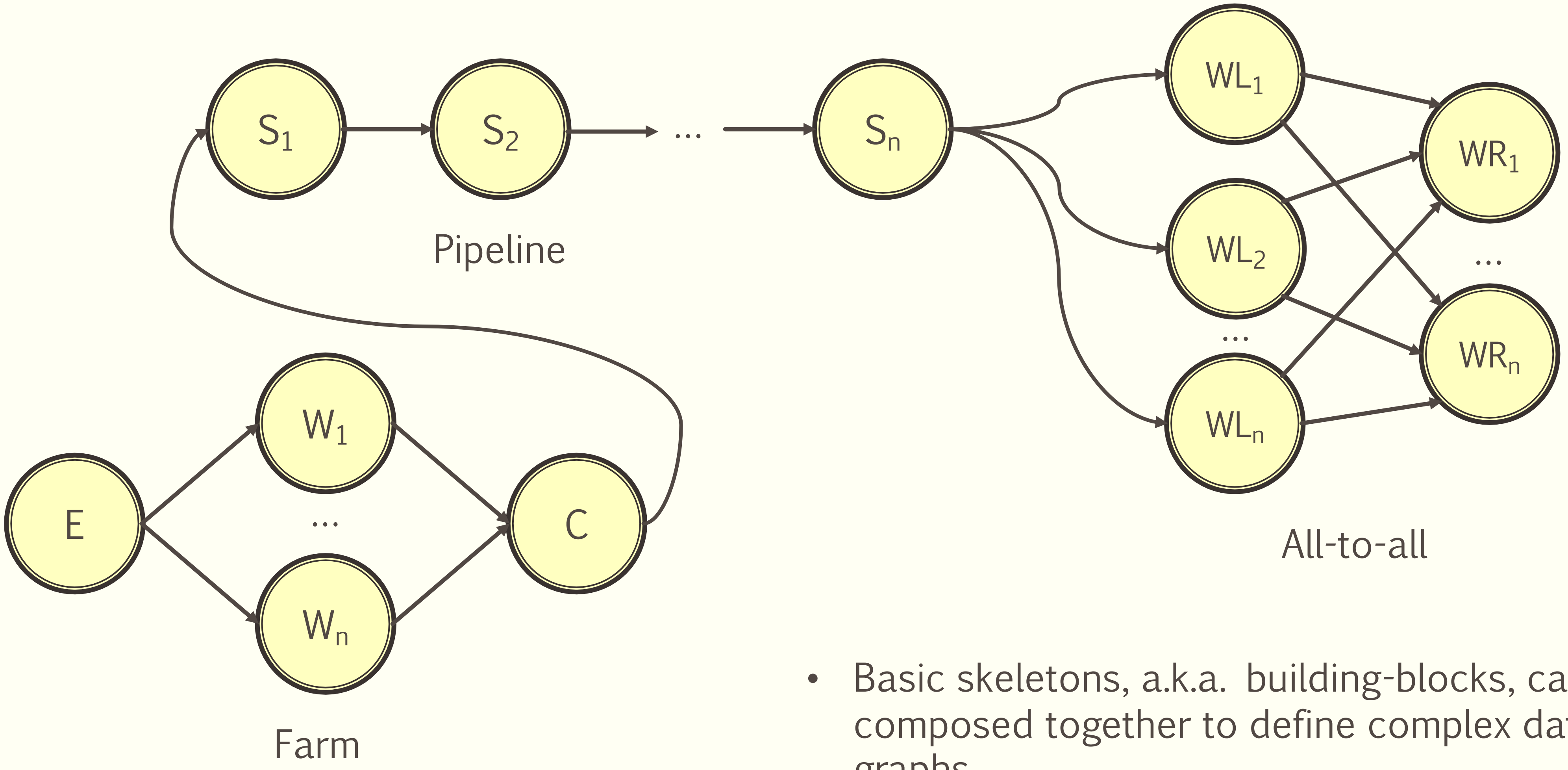
FastFlow

- Advocates high-level, pattern-based parallel programming.
- Mainly targets fine-grained and streaming applications.
- Originally designed for shared-cache multi-core
- Skeleton-based parallel programming model
- Support for hardware accelerators





FastFlow cont’d



- Basic skeletons, a.k.a. building-blocks, can be composed together to define complex data-flow graphs

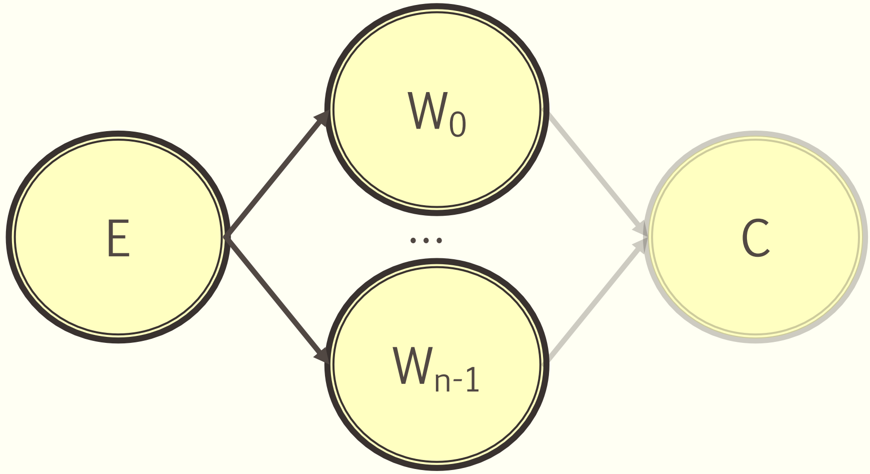
Programming model behind *FastFlow*

```
void Emitter () {
  for (i=0; i<streamLen; ++i){
     queue=SELECT_WORKER_QUEUE();
     queue->PUSH(create_task());
  }
}

void Worker() {
  while (!end_of_stream){
     myqueue->POP(&task);
     do_work(task);
  }
}

int main(){
  spawn_thread(Emitter);
  for (i=0; i<nworkers; ++i){
     spawn_thread(Worker);
  }
  wait_end();
}
```

Farm (functional replication) of n workers



From single to many multi-core machines

Need to scale

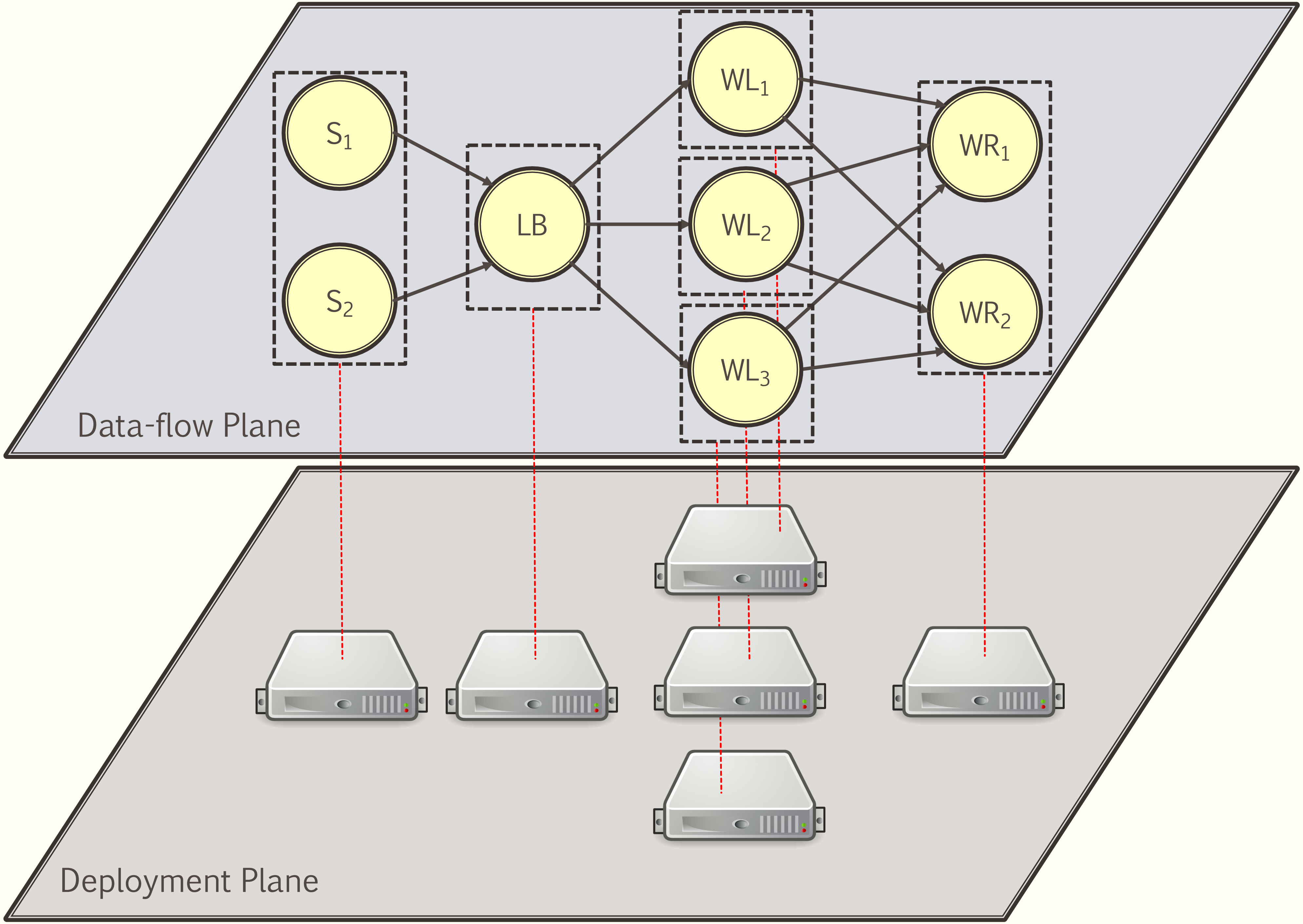
- Modern applications require to scale to hundreds/thousands to cores

Extending the FastFlow RTS

- We need to extend the *FastFlow* run-time system to work outside the single machine environment

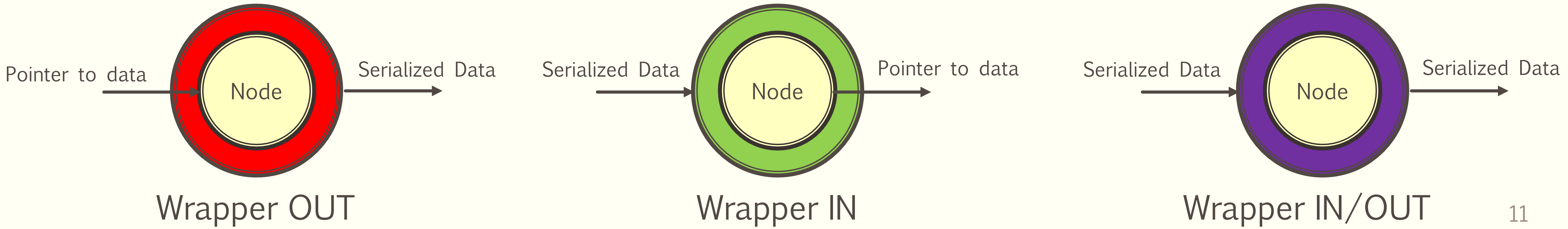
Goals

- Small effort by the programmers
- Use the same programming model of the shared-memory applications
- Lightweight layer reusing the built-in building-blocks


S1
S2
LB
WL1
WL2
WL3
WR1
WR2
Data-flow Plane
Deployment Plane

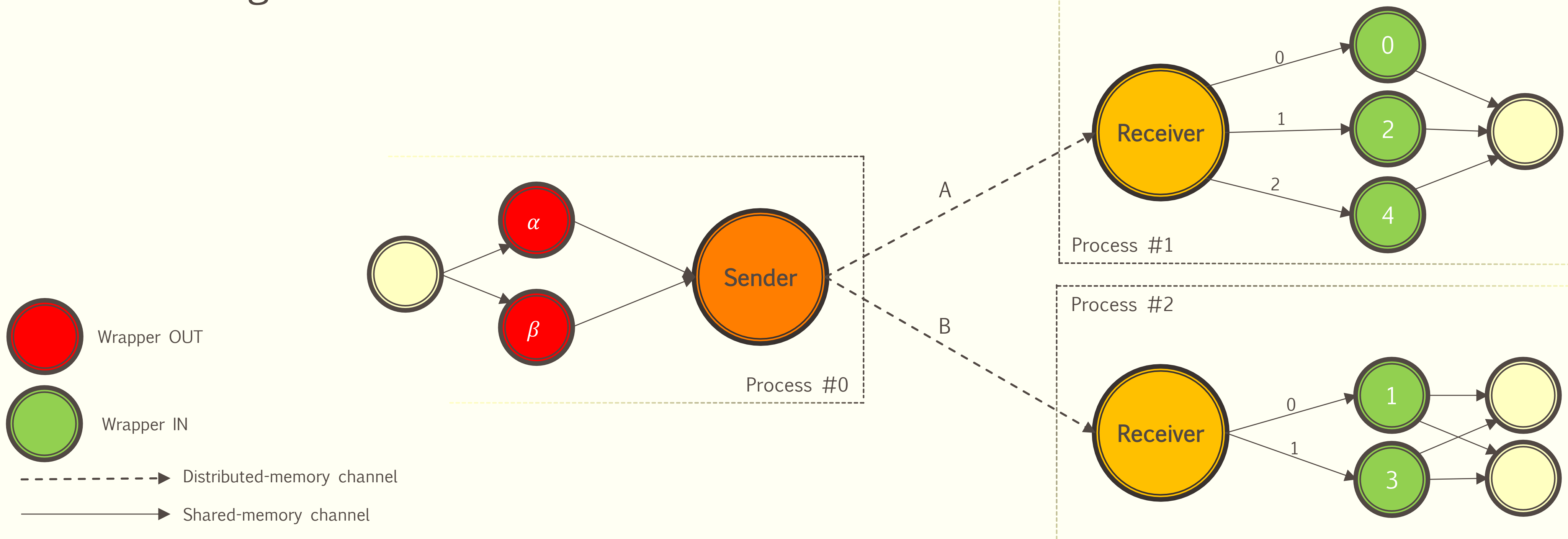
Communication wrappers

- Applied to edge-nodes, *i.e.*, those that communicate outside the process
- Do not require to reimplement edge-nodes
- Each edge-node can perform data serialization/deserialization in parallel. Two mechanism of serialization are available: *cereal* based, and user defined.
- Three types of wrappers
  - Wrapper OUT: performs data serialization and the encapsulation
  - Wrapper IN: performs the decapsulation and the data de-serialization
  - Wrapper IN/OUT: combination of Wrapper IN and Wrapper OUT for operators connected to remote nodes either in input and output.



Communication nodes

- **Sender** and **receiver** are the only nodes that actually communicate outside of the process (gateways).
- Lightweight routing protocol to during the initial handshake to perform message routing



Application Program Interface

myapp.cpp

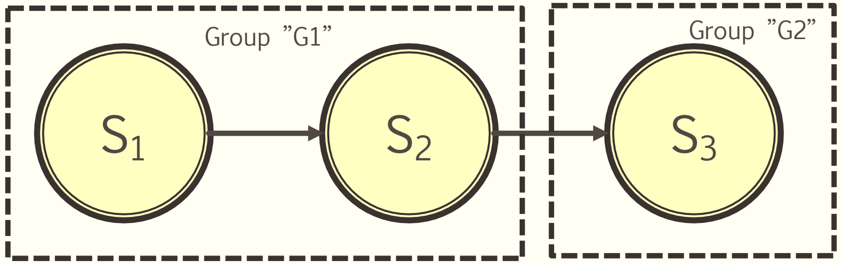
```cpp
#include <ff/ff.hpp>
#include <ff/dff.hpp>
...
int main(int argc, char*argv[]){
  DFF_Init(argc, argv);

  ff_pipeline pipe;
  Node1 s1;
  Node2 s2;
  Node3 s3;

  pipe.add_stage(&s1);
  pipe.add_stage(&s2);
  pipe.add_stage(&s3);

  auto G1 = pipe.createGroup("G1");
  auto G2 = pipe.createGroup("G2");
  G1 << &s1 << &s2;
  G2 << &s3;

  pipe.run_and_wait_end();
  return 0;
}
```



config.json

```json
{
  "groups" : [
      {
       "endpoint" : "host1:0",
       "name" : "G1",
      },
      {
       "name" : "G2",
       "endpoint": "host2:8005",
      }
    ]
}
```

```
host1:$ ./myapp --DFF_Config=config.json --DFF_GName=G1
```

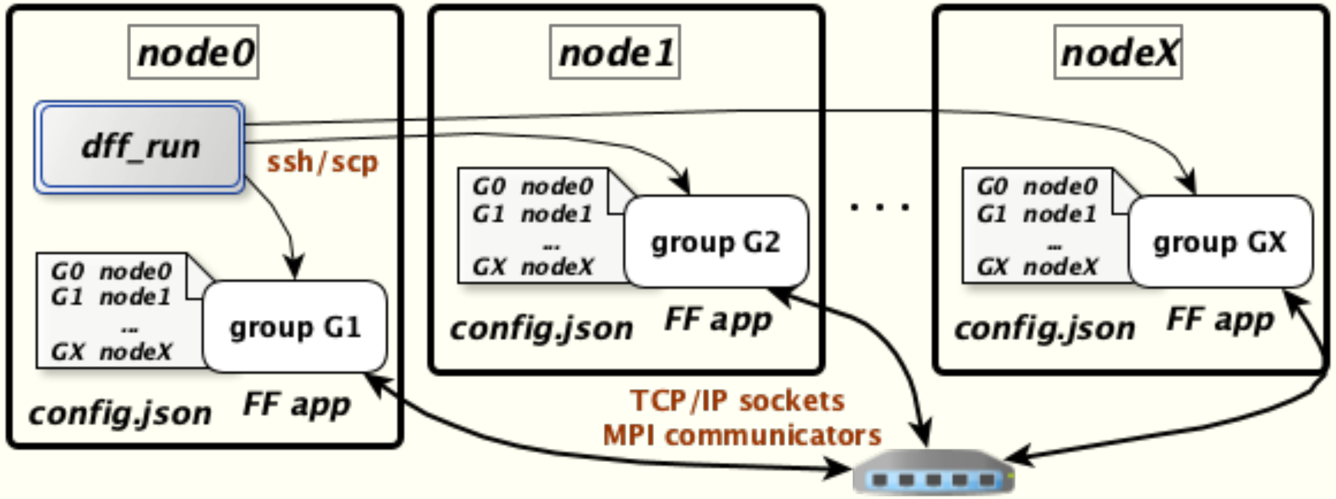
```
host2:$ ./myapp --DFF_Config=config.json --DFF_GName=G2
```

Alternatively

```
laptop:$ dff_run -V -f config.json ./myapp
```

dff_run loader module

- Like the well-known *mpirun* command
- Takes in input the executable and the JSON configuration file
- Forks the processes either locally and remotely, with the right parameters



- Still as proof of concept.
- Mainly used to accelerate the launching during troubleshooting and test execution.

Open problems and future directions

- Automatic graph split
- Bridging with Cloud/Big Data applications
- Collective communications (e.g., broadcasts, gathers all)
- Add fault-tolerance features

TARGETING SCALE-OUT AND SCALE-UP PLATFORMS WITH A UNIFIED PARALLEL PROGRAMMING MODEL

Thanks for the attention!

Nicolò Tonci
nicolo.tonci@phd.unipi.it