Introduction to PROOF and PROOF Lite

ROOT Training at IRMM
1st March 2013

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

Does the load on your 8-core machine look like this during your analysis session?

What would be needed to make it look like this?

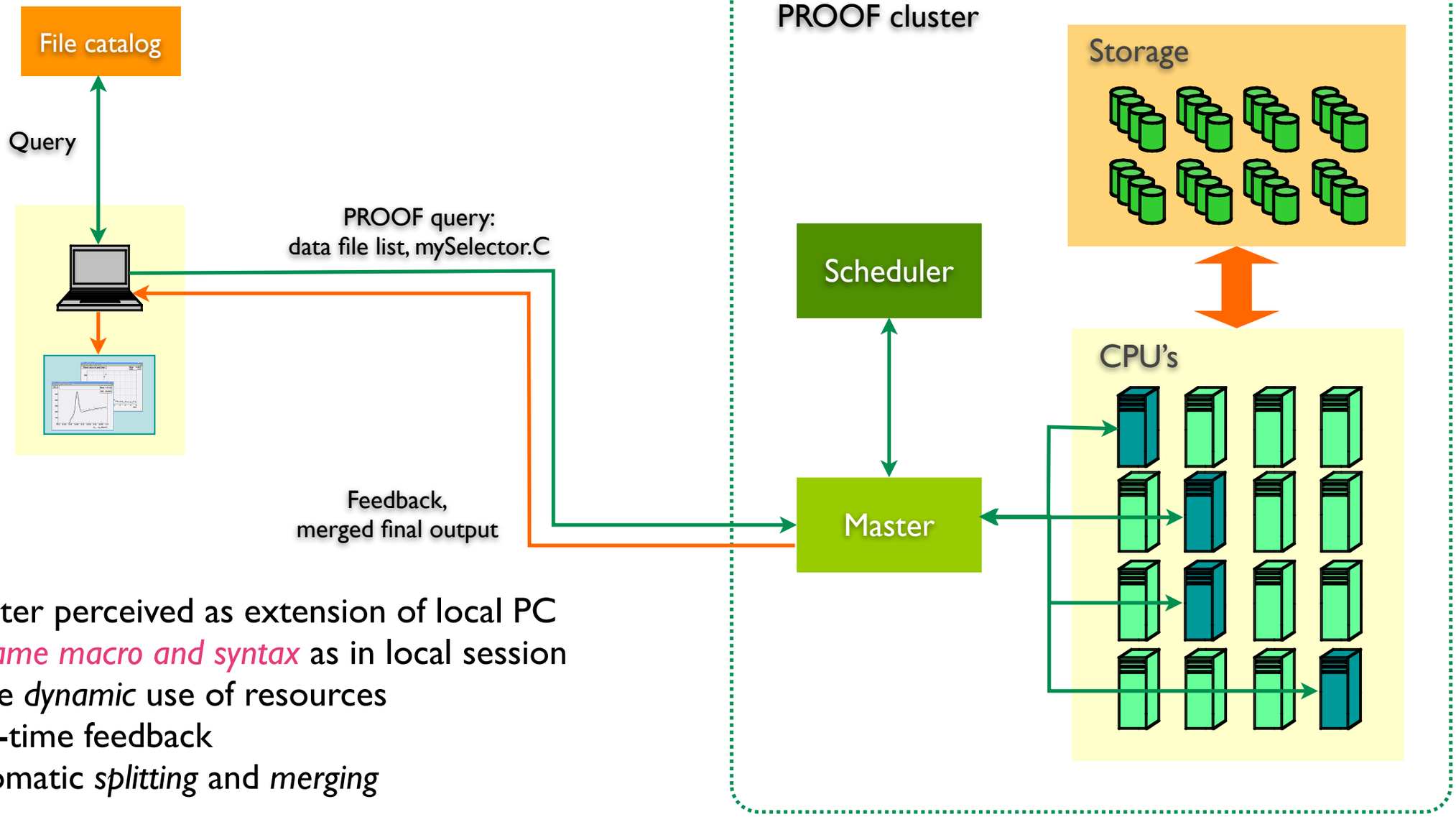
Just one extra command

TProof::Open(“”)

What is PROOF?

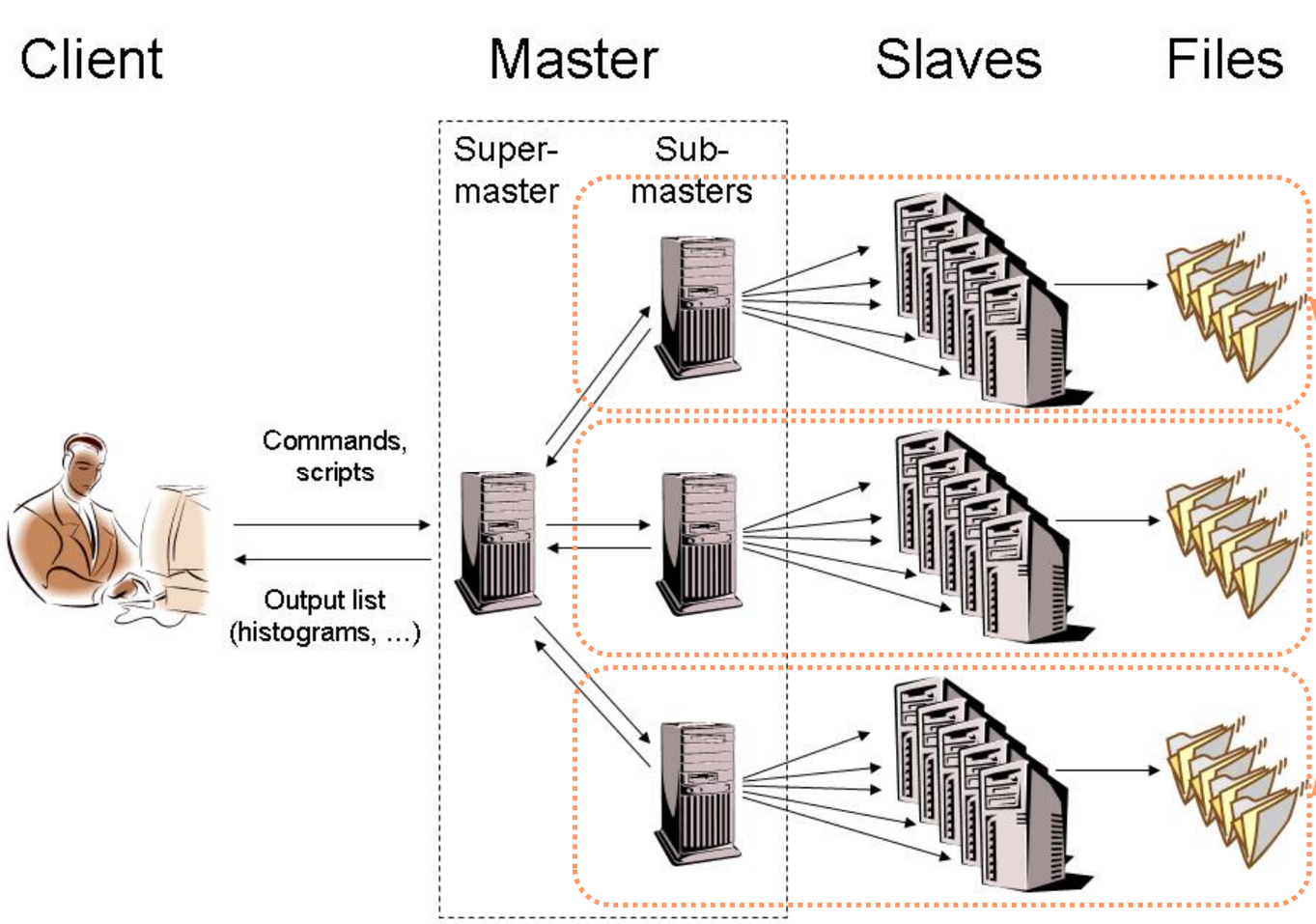
- A system for running ROOT queries in parallel on a large number of distributed computers or many-core machines.
- PROOF is designed to be a transparent, scalable and adaptable extension of the local interactive ROOT analysis session.
- Extends the interactive model to long running “interactive batch” queries.
- Uses xrootd for data access and communication infrastructure.
- For optimal CPU load it needs fast data access (SSD, disk, network) as queries are often I/O bound.
- Can also be used for pure CPU bound tasks like toy Monte Carlo’s for systematic studies or complex fits.

The PROOF Approach



- Cluster perceived as extension of local PC
  - *Same macro and syntax* as in local session
- More *dynamic* use of resources
- Real-time feedback
- Automatic *splitting* and *merging*

Multi-Tier Architecture



Adapts to wide area *virtual* clusters

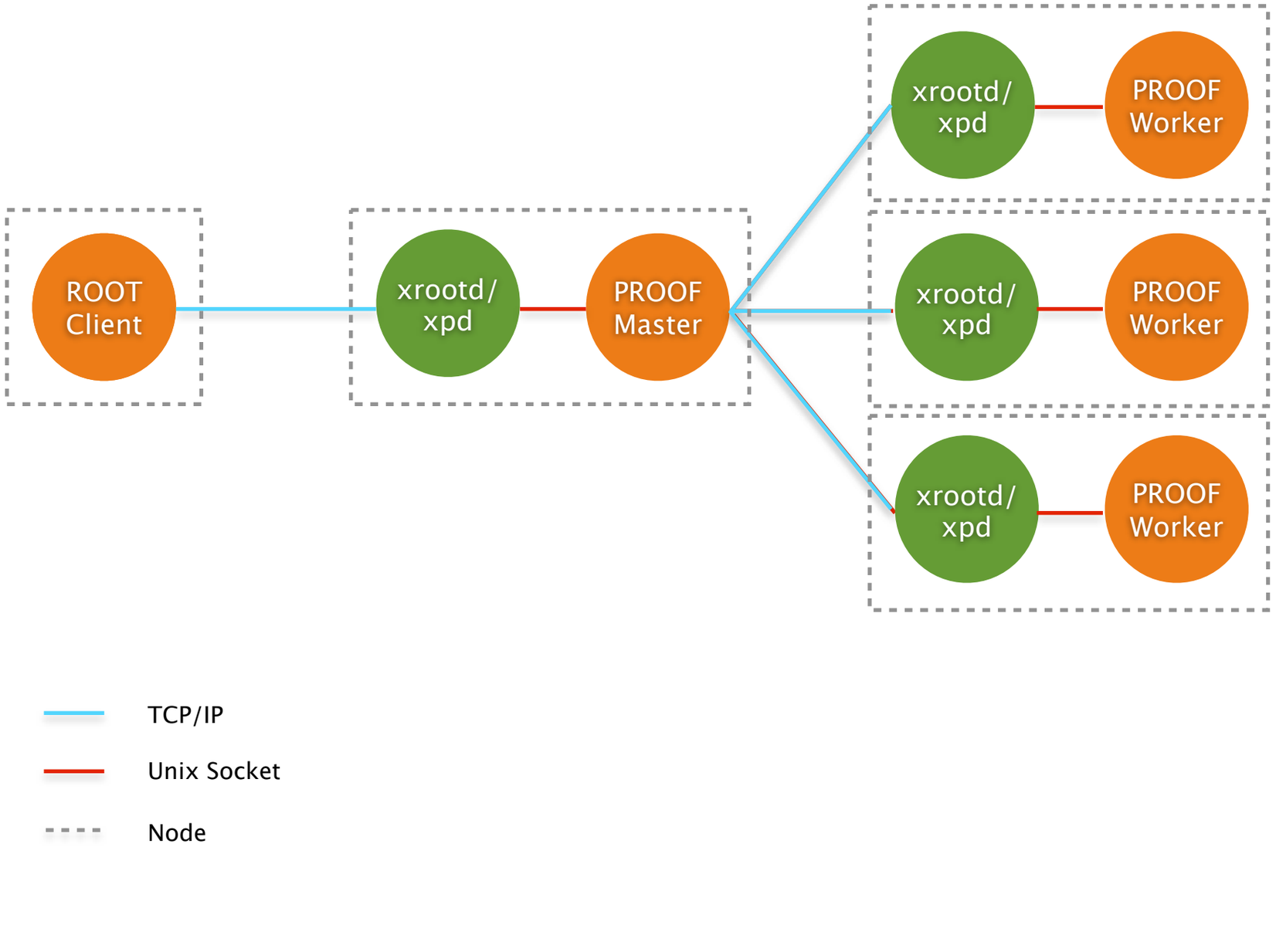
Geographically separated domains, heterogeneous machines

Network performance

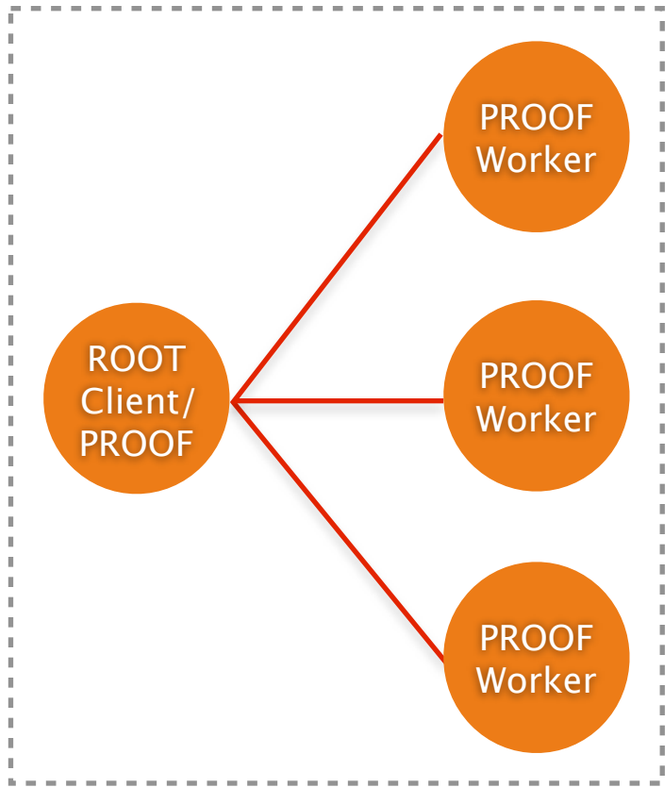
Less important → VERY important

Optimize for data locality or high bandwidth data server access

From PROOF


ROOT Client
xrootd/ xpd
PROOF Master
xrootd/ xpd
PROOF Worker
xrootd/ xpd
PROOF Worker
xrootd/ xpd
PROOF Worker
TCP/IP
Unix Socket
Node

To PROOF Lite



Unix Socket

Node

What is PROOF Lite?

- PROOF optimized for single many-core machines.
- Zero configuration setup (no config files and no daemons).
- Workers are processes and not threads for added robustness.
- Like PROOF it can exploit fast disks, SSD's, lots of RAM, fast networks and fast CPU's.
- Once your analysis runs on PROOF Lite it will also run on PROOF.
- Works with exactly the same user code as PROOF.

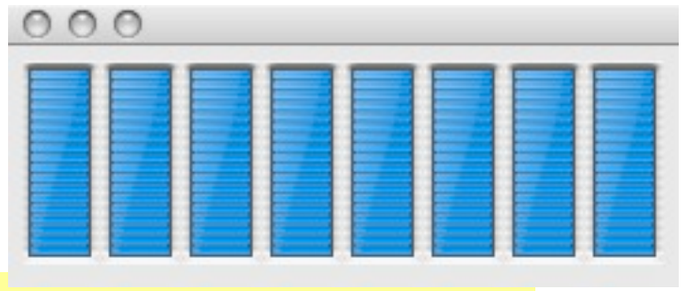
How Can I Use PROOF Lite

- Get rid of your own event loop.
- Coding your own event loop is error prone anyway.
- Use the `TSelector` framework.
- Let ROOT make the event loop, it knows how to do it
- Use properly split `TTree`'s for fast access
- Compile your code on-the-fly with ACliC

PROOF Lite Sample Session

```
void go_ana(int aclic = 1)
{
   gROOT->ProcessLine(".L makeChain.C");
   TChain *chain = makeChain();

   Long64_t evtmax=2619000;
   gROOT->Time();
   if (aclic) chain->Process("TSelector_Ntuple_Zee.C+","",evtmax);
   else       chain->Process("TSelector_Ntuple_Zee.C", "",evtmax);
}
```

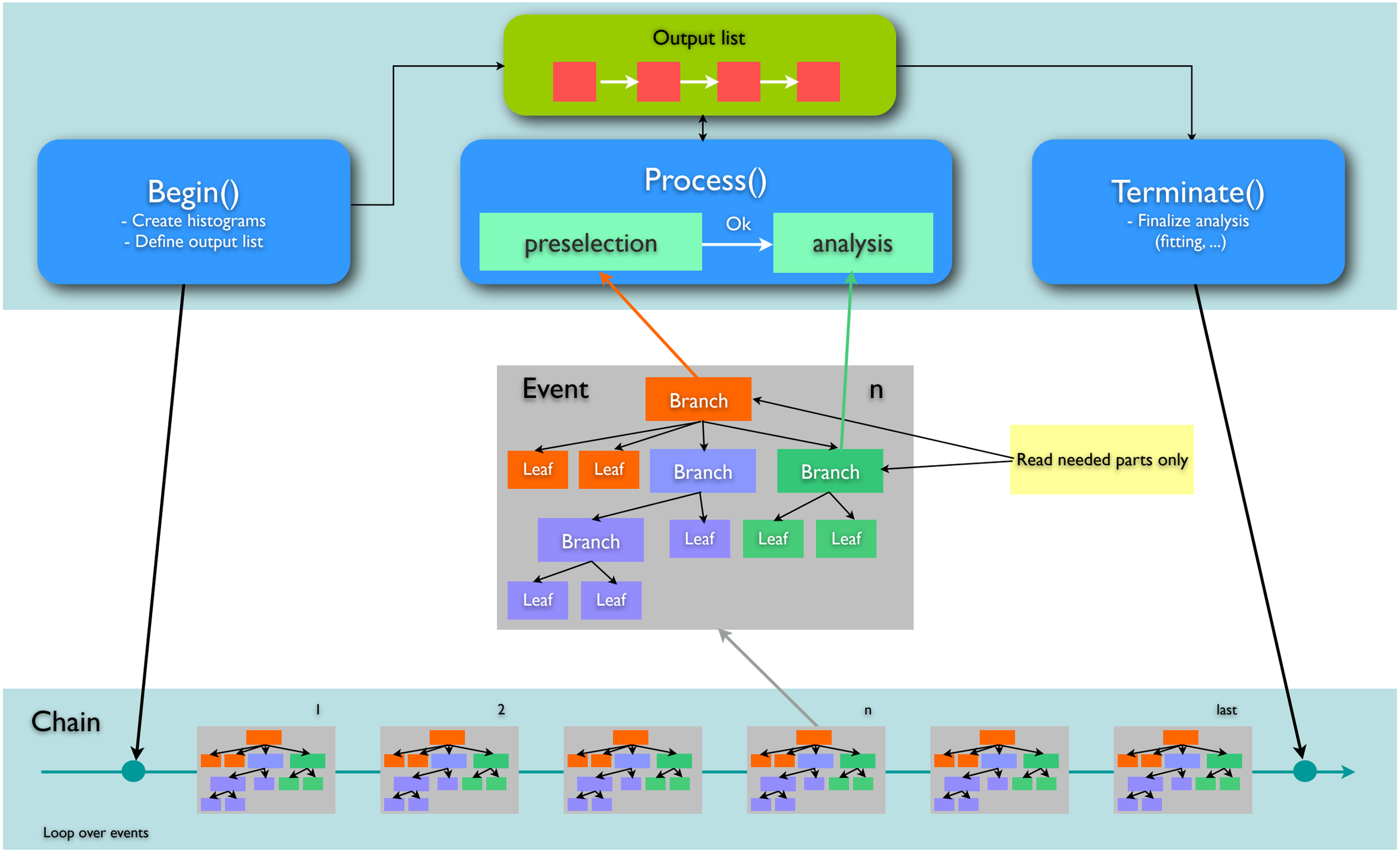


```
void go_ana(int aclic = 1)
{
   gROOT->ProcessLine(".L makeChain.C");
   TChain *chain = makeChain();

   TProof::Open("");
   chain->SetProof();

   Long64_t evtmax=2619000;
   gROOT->Time();
   if (aclic) chain->Process("TSelector_Ntuple_Zee.C+","",evtmax);
   else       chain->Process("TSelector_Ntuple_Zee.C", "",evtmax);
}
```

The ROOT Data Model
Trees & Selectors



TSelector – User Code

```
// Abbreviated version
class TSelector : public TObject {
protected:
    TList *fInput;
    TList *fOutput;
public
    void   Notify(TTree*);
    void   Begin(TTree*);
    void   SlaveBegin(TTree *);
    Bool_t Process(int entry);
    void   SlaveTerminate();
    void   Terminate();
};
```

TSelector::Process()

```
...
...
// select event
b_nlhk->GetEntry(entry);          if (nlhk[ik] <= 0.1)     return kFALSE;
b_nlhpi->GetEntry(entry);         if (nlhpi[ipi] <= 0.1)   return kFALSE;
b_ipis->GetEntry(entry); ipis--; if (nlhpi[ipis] <= 0.1) return kFALSE;
b_njets->GetEntry(entry);         if (njets < 1)           return kFALSE;

// selection made, now analyze event
b_dm_d->GetEntry(entry);          //read branch holding dm_d
b_rpd0_t->GetEntry(entry);        //read branch holding rpd0_t
b_ptd0_d->GetEntry(entry);        //read branch holding ptd0_d

//fill some histograms
hdmd->Fill(dm_d);
h2->Fill(dm_d,rpd0_t/0.029979*1.8646/ptd0_d);
...
...
```

The Packetizer

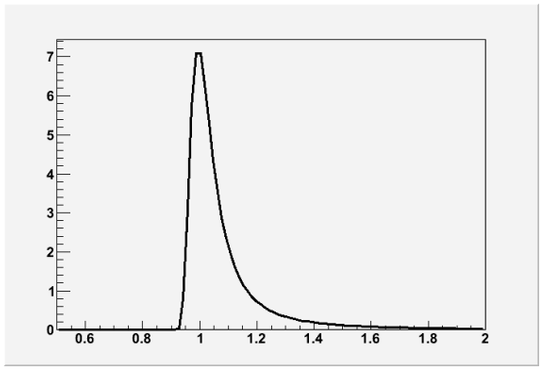
- The packetizer is the heart of the system
- It runs on the client/master and hands out work to the workers
- The packetizer takes data locality and storage type into account
- Tries to avoid storage device overload
- It makes sure all workers end at the same time

Pull architecture

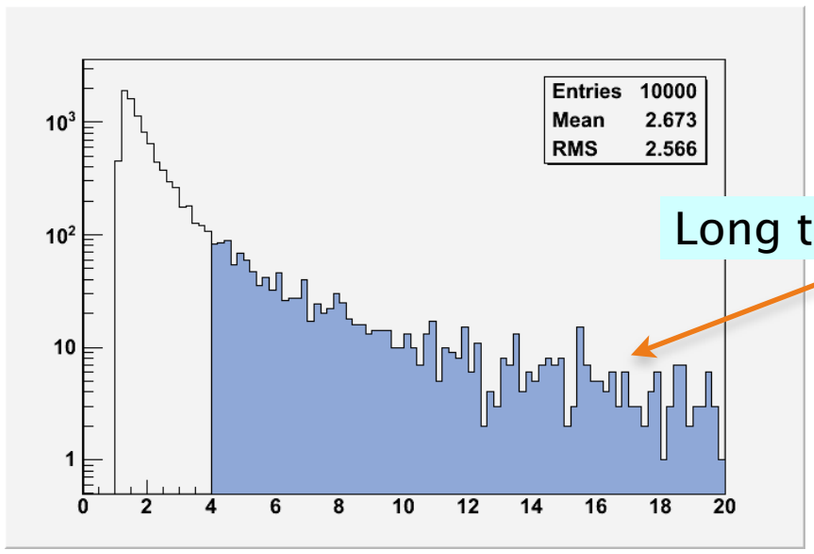
workers ask for work, no complex worker state in the master

PROOF Pull Technology Avoids Long Tails

- In push approach last job determines the total execution time
- Basically a Landau distribution
- Example:
- Total expected time 20h, target 1h
- 20 sub-jobs, 1h +/- 5%



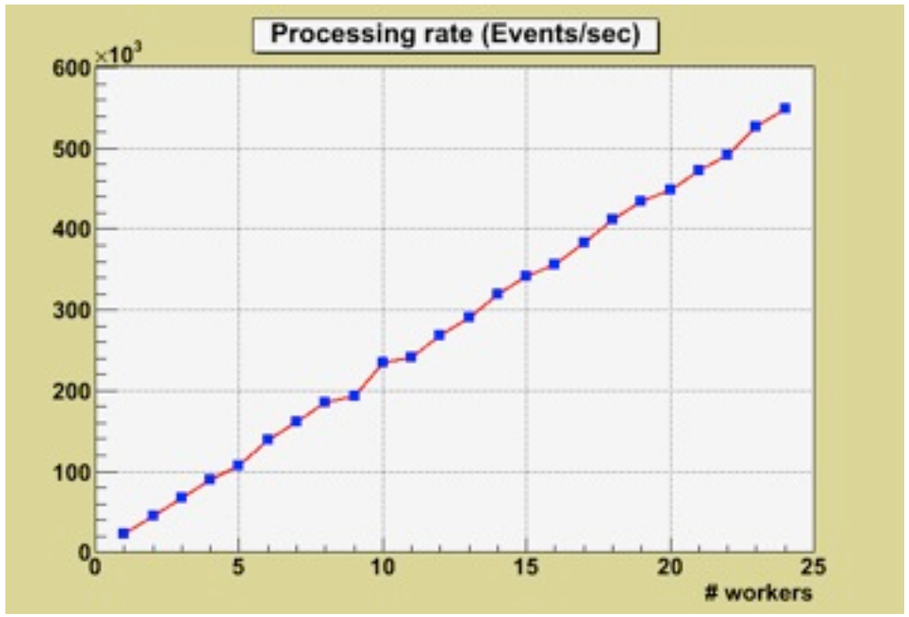
10000 toy experiments

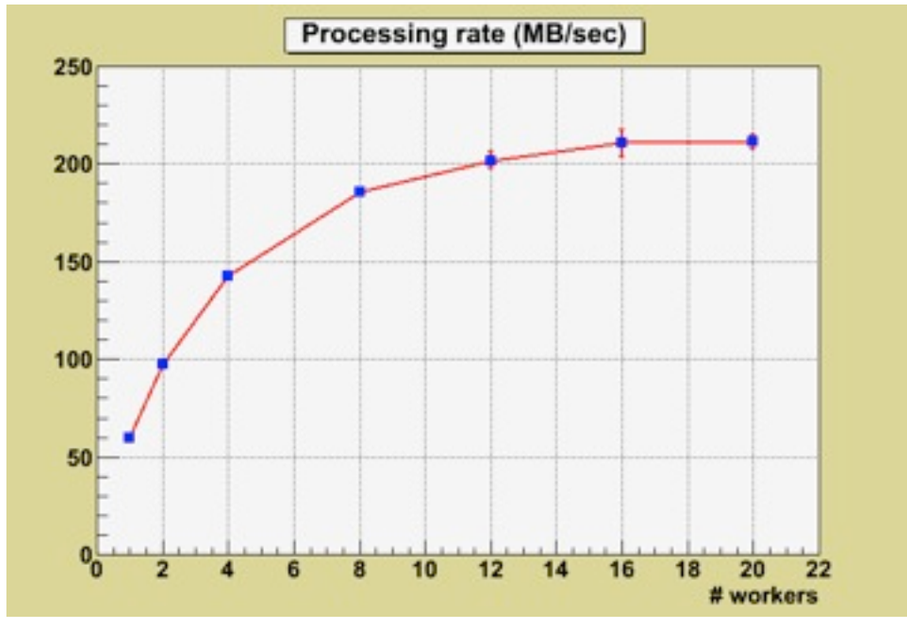


Long tails, e.g. 15% > 4h

Time of slowest sub-job

PROOF Lite Performance





SSD's 10x times faster than HDD in concurrent read