Achieving real time response in grid applications.

Stefano Cozzini,
Riccardo Di Meo

The grid

- The actual implementation is queue and not real time oriented: this is definitely an issue for many dynamic applications.
- Our main goal is to attain real time response from grid enabled applications.

Obstacles that should be overcome.

- The queue approach is completely inadequate for real-time tasks: we don't know when our program will be executed.

The program should be ready to immediately accept requests.

- Input files need to be downloaded from the SE: this takes time.
- All steps, from submission to results retrieval, add a significant delay, which is unavoidable as long as standard tools are used.

The total time needed to submit a request and obtain an answer should be as small as possible.

- The execution is not interactive: after sending the job to the RB, there's no way to alter it.

A single job should be able to process many requests.

...and solutions.

- The queue approach is completely inadequate for real-time tasks: we don't know when our program will be executed.
- Input files need to be downloaded from the SE: this takes time.
- All steps, from submission to results retrieval, add a significant delay, which is unavoidable as long as standard tools are used.
- The execution is not interactive: after sending the job to the RB, there's no way to alter it.

We book resources in advance in order to have enough at a given time ("*Job reservation*").

The WN downloads the data while waiting.

We bypass the information system, obtaining status and results directly from the WN.

We establish a direct connection between WN and UI, thus letting them interact.

Job reservation

- We submit many requests in advance in order to have resources ready when needed: once each job is running, it waits until the user has some data to process.

- No outside host can establish a connection to the WN since they are on private networks:
  - we need a reversed approach: the WN itself polls periodically an external host (which must be resolvable).
  - we need outbound connectivity for the WNs (which is sometimes blocked for security reasons), EGEE is working on a facility that opens ports on demand.
  - although this approach works, it is more complicated to code than the (more intuitive) one where the UI is the client that contacts the WN on the grid.

Some problems
(and suggestions to overcome them...)

- Doesn't scale very well and is only statistically accurate (you cannot be absolutely sure that the resources will be there).
  - There's no simple solution to this problem: the limit lies in the actual implementation and can be partially bypassed through a more responsive queue, although this problem is *intrinsic in every shared system* where N+1 users compete for N resources...
  - A statistical analysis can be done to maximize the *chance* of having free resources.

More problems
(and suggestions as well...)

- Advanced booking wastes resources and could be too expensive.
  - the “Wall Time” and not the “CPU Time” is what is usually accounted.
- This issue can be easily resolved by running 2 programs per job:
  - the job reserving one, that will simply contact the UI to check if there are jobs/transactions to do.
  - a time consuming one (in the background with a low priority), which effectively uses all the CPU time wasted by the first one, which should:
    - checkpoint often and efficiently.
    - be embarrassingly parallel.

```
#!/bin/sh

chmod u+x ./filler_app
nice -n 10 filler_app &

chmod polling_app
./polling_app
myhost.mynet.edu
```

Executed at low priority for avoiding interference with the real time application. Uses the otherwise wasted CPU time.

Poll the UI for tasks: will take over when needed.

```
Executable = the_script_on_the left;
InputSandbox =
{"filler_app","polling_app",
     the_script_on_the_left};
OutputSandbox =
{"filler_app_results.dat"};
```

To effectively apply this approach the filler application should be able to quickly and efficiently checkpoint.

Special care must be taken in multi processor/core systems to avoid stealing resources from other users!!!

Real time response.

- Now that we have our resources available when we really need them, some problems still remain:
  - Every interaction with the grid brings a delay which cannot be tolerated for real time (or *near* real time) applications.
    - This overhead increases with the load of the RB.
    - The RB cannot be replicated in a transparent way.
    - The delay is significant no matter the load.

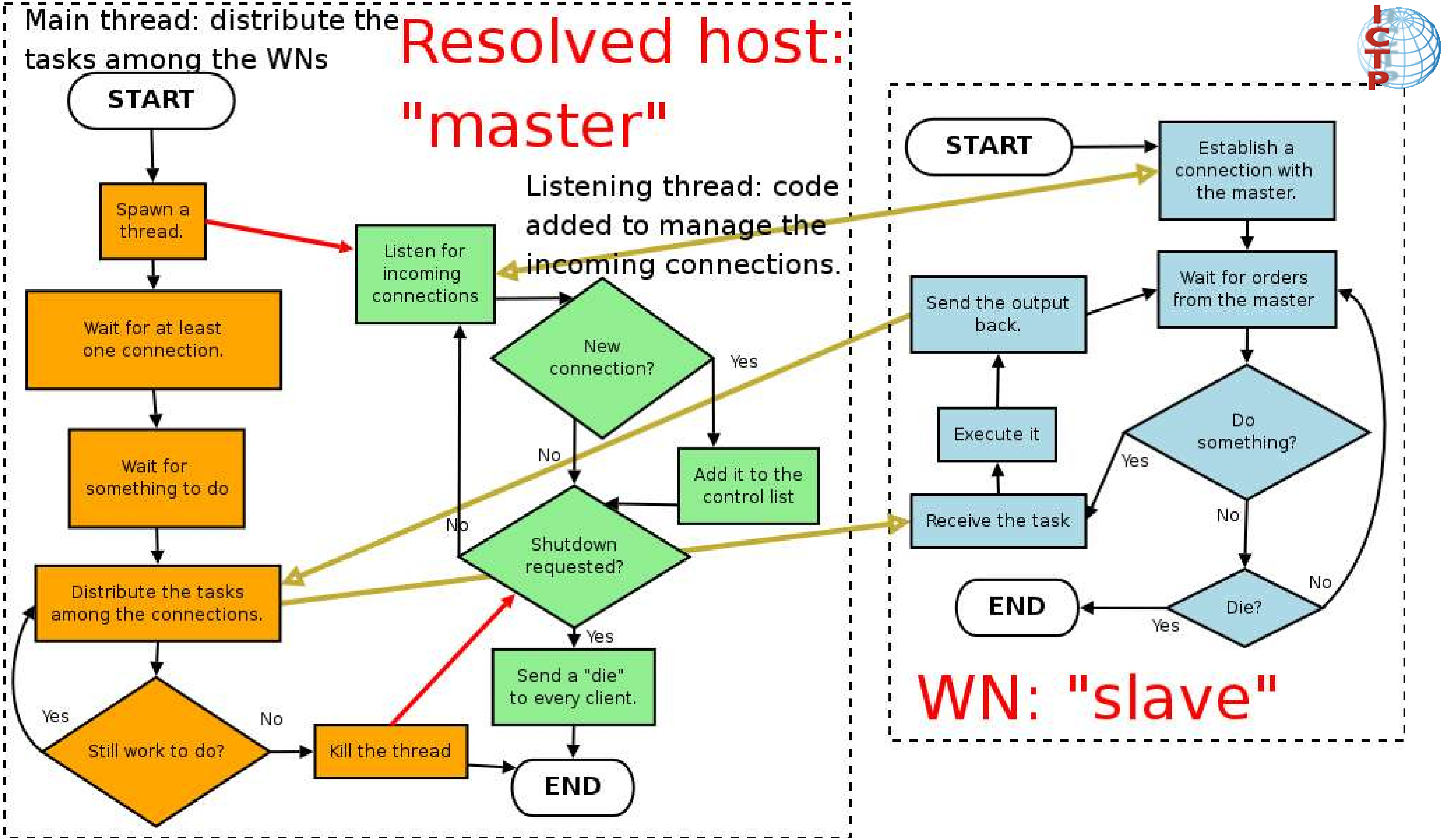
The standard workflow cannot be applied.

How?

- The most straightforward way is to directly communicate with our job on the WNs.

- With this paradigm we don't need anymore:
  - a status command
    - The establishment of a connection is the signal that the app. is runing, after that, every information is received directly.
  - a get-output command
    - Files can be sent to the user without passing through the RB, with the added benefit that the load on it decreases.
  - a cancel command and different jobs for every single task
    - The application can run on the WN and satisfy multiple requests until we explicitly tell it to quit.

- An “interactive” JobType exists, which creates a pipe between WN and UI, although in this way:
  - **the WNs can connect to the UI only**.
  - a lot of overhead: allocates many resources and scales badly.
  - every job allocates a different port and creates 2 special files: the listening application has to be aware of the details of the submission (at least the id list of jobs) to parse the correct ones!!
  - conflicts with the job reservation.
  - we have no control over the type/number of streams.
  - we'll have to use sockets anyway (see later) so... why bother?
- the “interactive” job, though easy to use and fine for single submissions (which is its original purpose) is **not well suited for our needs:** plain sockets is the answer.



Main thread: distribute the tasks among the WNs

Resolved host: "master"

START

Spawn a thread.

Wait for at least one connection.

Wait for something to do

Distribute the tasks among the connections.

Still work to do?

Yes

No

Kill the thread

END

Listening thread: code added to manage the incoming connections.

Listen for incoming connections

New connection?

Yes

No

Add it to the control list

Shutdown requested?

No

Yes

Send a "die" to every client.

END

WN: "slave"

START

Establish a connection with the master.

Wait for orders from the master

Do something?

Yes

No

Receive the task

Execute it

Send the output back.

Die?

No

Yes

END

This model is very suitable for a portal-like application were different types of real-time apps. get submitted through a custom interface on the UI.

But life is tough, and sometimes....

- the computing power of single WN is not sufficient.
- the tasks are not independent.

Parallelization is needed.

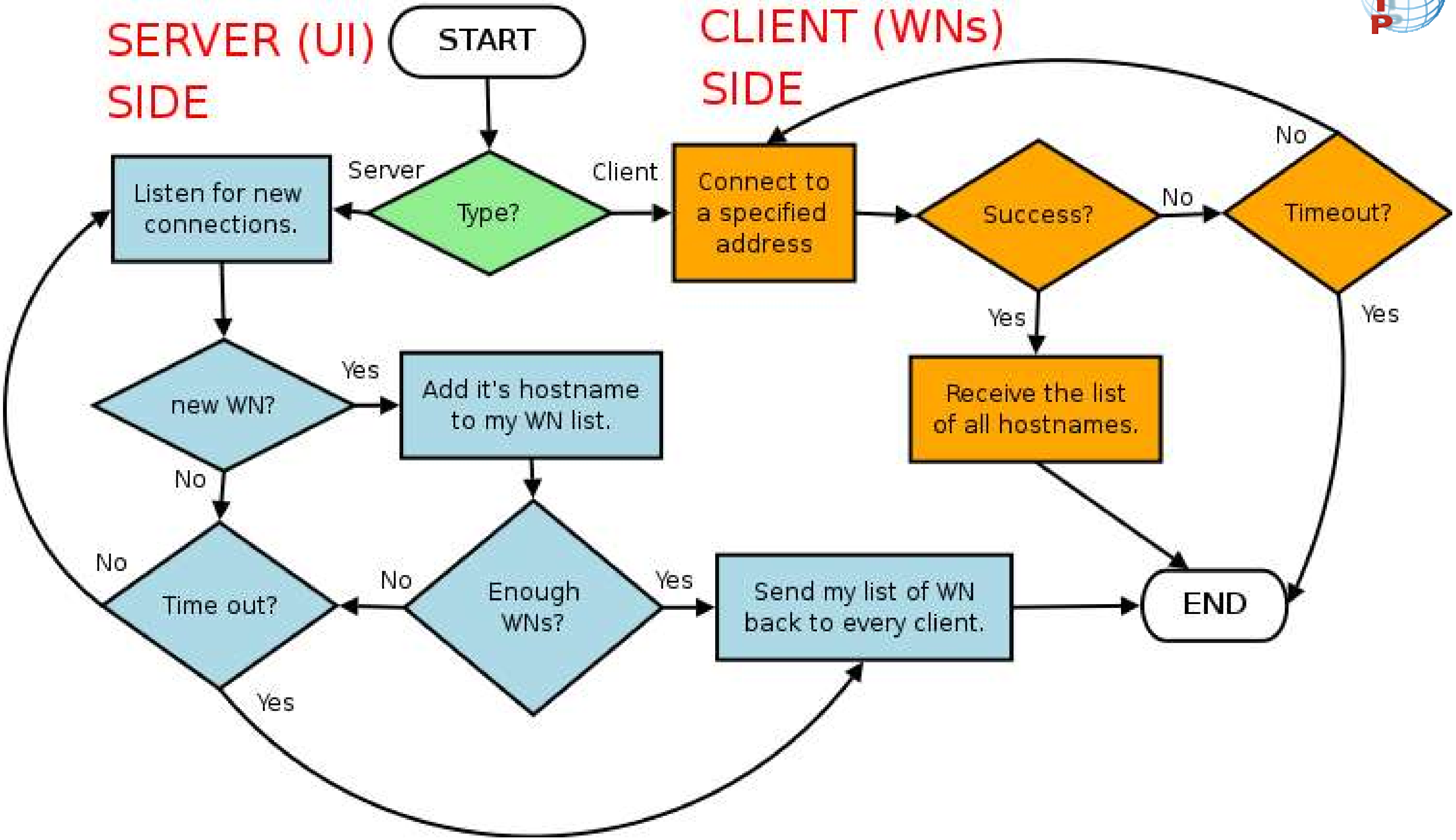
- MPI cannot be used since it conflicts with Job Reservation:
  - during J.R. you ask for many WN to get only some of them asynchronously.
  - with MPI every WN should be present at once: it's an “all or nothing” approach.

We need intra cluster communication.

- In the previous scenario the link between WNs can be provided by the master: input data travels through a public network: this approach is slow: WNs should rely on the fast cluster net!
- Aren't we forgetting something?
  - WNs coming from independent jobs are not aware of each other (they don't know the other's IP).
  - no MPI implies sockets again.

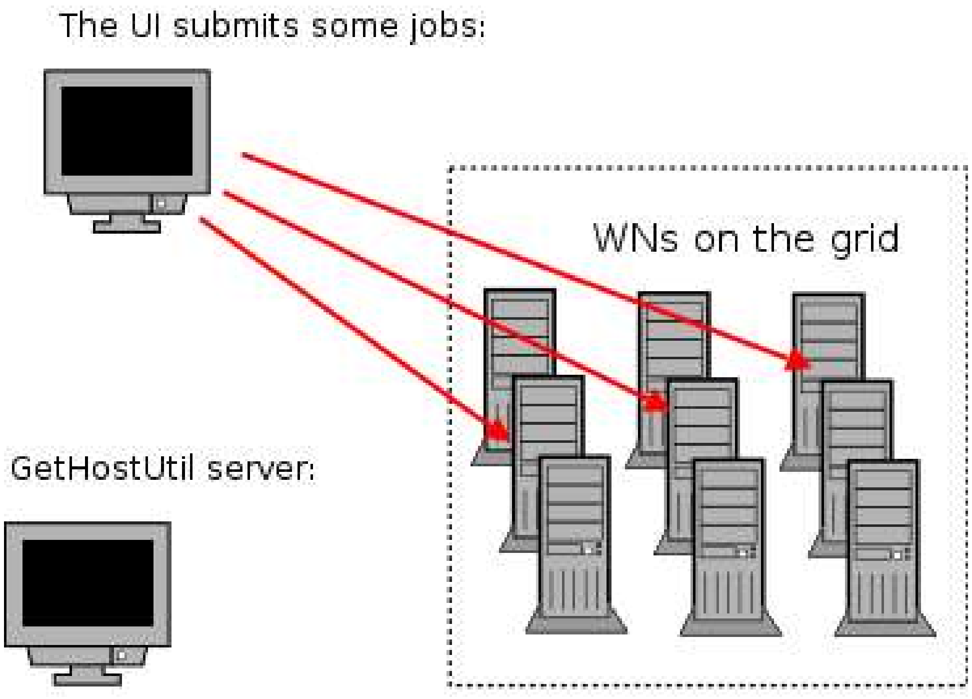
GridHostUtil

- We developed a small utility (gridhostutil) that:
  - has a client/server option:
    - the server listens on the UI (or other resolved host) for incoming connections.
    - the client is run on the WNs and connect to the server.
    - After enough connections are received or a timeout elapses, the server contacts the clients and sends them their complete and ordered list.
  - allow WNs inside a single CE to communicate.


SERVER (UI) SIDE
START
CLIENT (WNs) SIDE
Server
Type?
Client
Listen for new connections.
Connect to a specified address
Success?
No
Timeout?
No
Yes
Yes
Receive the list of all hostnames.
new WN?
Yes
Add it's hostname to my WN list.
No
No
Time out?
No
Enough WNs?
Yes
Send my list of WN back to every client.
END
Yes

A possible implementation

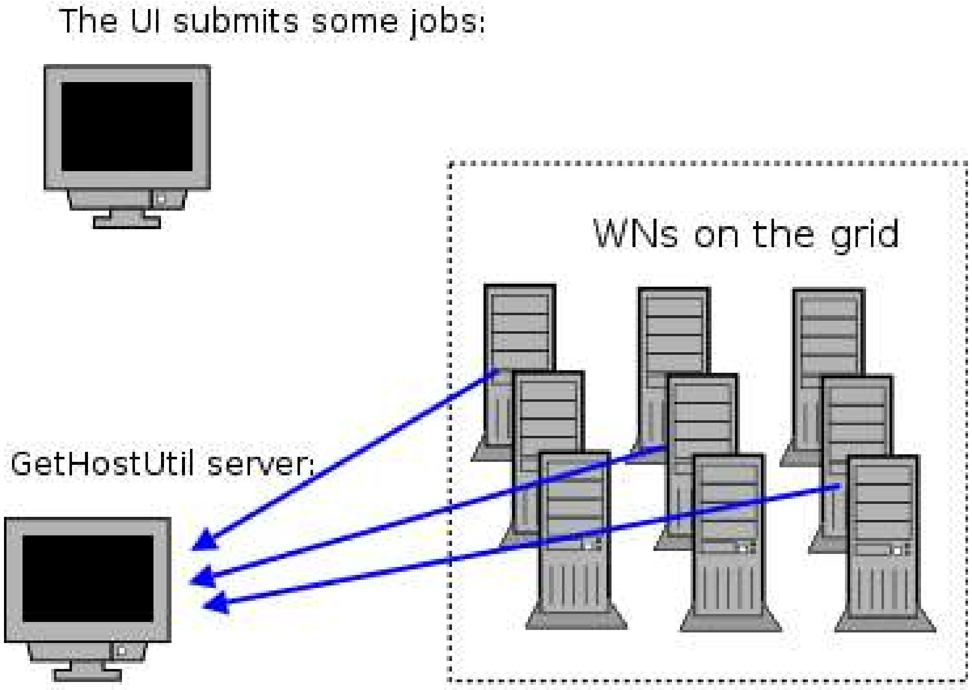
- After some jobs are launched on the grid, a server is executed on a resolved host.



•The middleware is needed only during this step.
•The jobs are submitted in advance and are ready when users need them.
•While waiting, the WNs work on a time consuming job.
•GridHostUtil is launched on them as a client.

A possible implementation

- The Hostnames exchange begins.



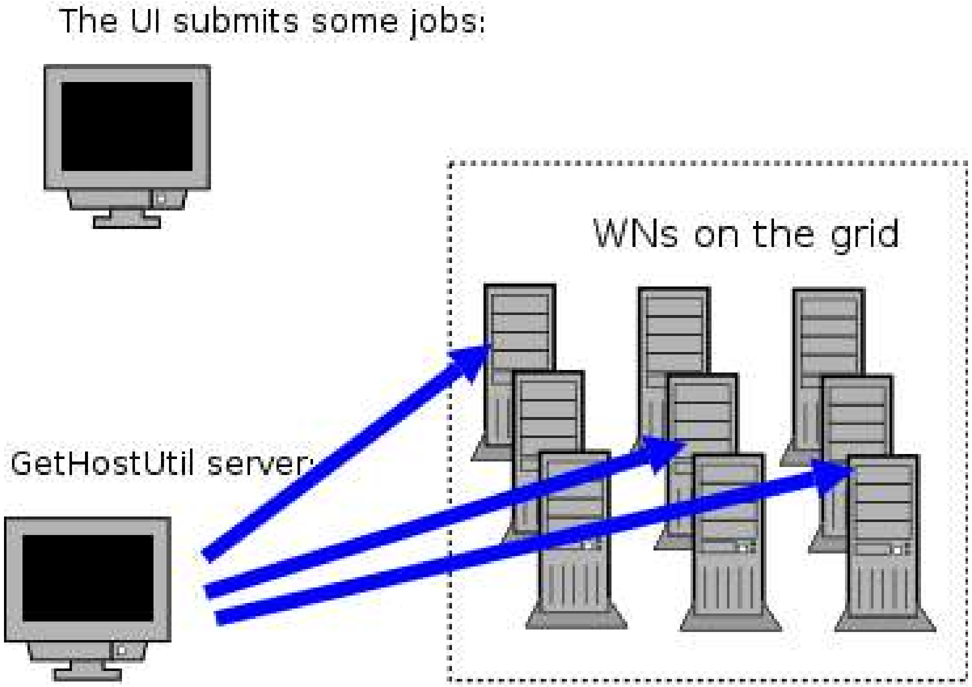
•The GridHostUtil client tries repeatedly to contact a resolved host.

•The host that has to be called is known in advance.

•Up to this point, the WNs are completely separated.

A possible implementation

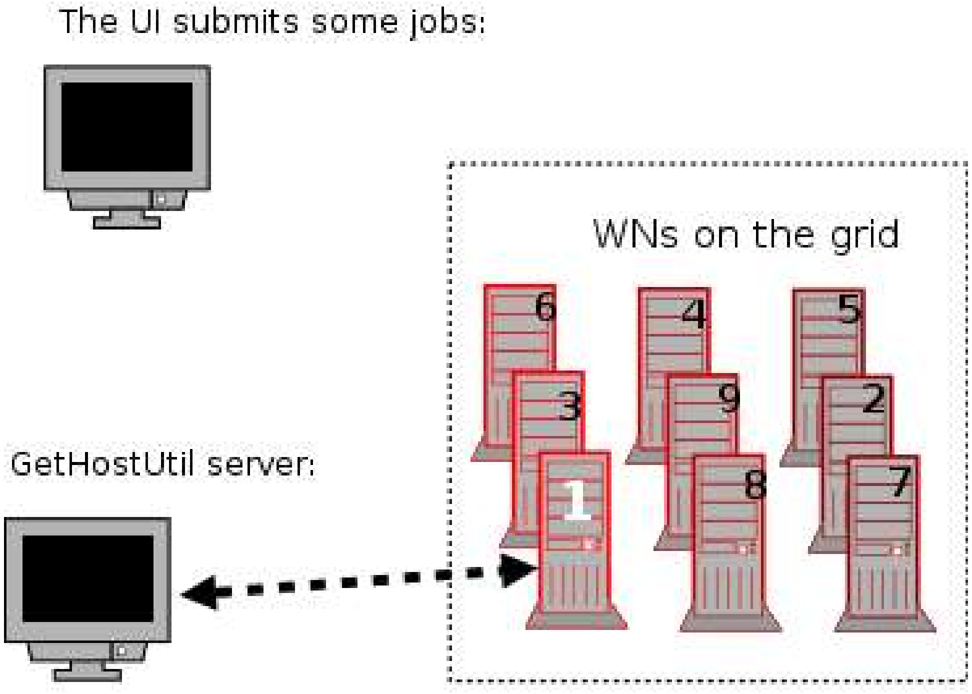
- The resolved host becomes aware of the number of available hosts on the grid.



•The GridHostUtil server waits until enough WNs have contacted it.
•A timeout can be specified.
•Once one of those conditions is satisfied, it sends a complete and ordered list of hostnames to all Wns.
•Lists are slightly different for each WN..

A possible implementation

- The WNs are now aware of each other and can communicate freely.



•A server on an external host can execute applications remotely in real time, thus acting as a “queue under the middleware's queue”.

•Though not implemented, nothing prevents recruiting other WNs: at this point (by repeating the previous steps), if the application is able to handle them.

Future directions

- We already explored inter-CE communication between WNs with the help of a bridging host and it works for small amounts of data.
- We are heading for the development of an MPI like library to simplify the porting of real time applications on the grid without having to work directly with sockets.
- The opportunities offered by XML-RPC and ssh tunneling might be worth exploring.

Conclusions

- Real time response from the grid is achievable, although some effort is required to port the application (unless it's using sockets already, in which case the porting is straightforward).
  - Job reservation can be used to get a ready WN when needed.
  - The reversed connection approach provides the interactivity and prompt answer from the grid.

Thank you!

Questions?