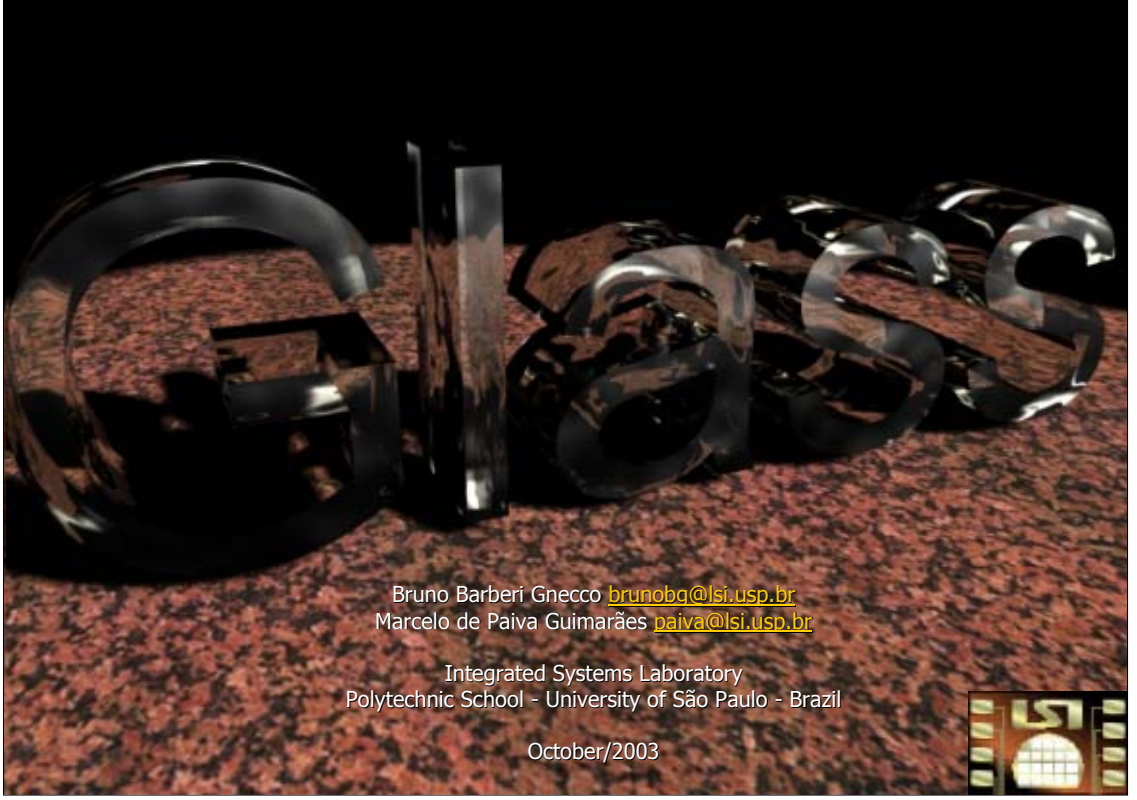


This presentation introduces Glass.

Glass is a new library for distributed computing. It was designed to fulfill a number of requirements:

•Extensibility and flexibility: the library must the easy to extend, without requiring API changes or even recompilation. This way, it will be always up-to-date with the latest technologies and be able to solve specific needs of users.

•Portable and interoperable: distributed computing is everyday more heterogeneous. The library must not only run in different architectures and operating systems, but interoperate among them.

•Easy to use, fast learning curve: distributed APIs tend to be complicated, requiring a lot of time to understand and master. Most libraries provide a huge number of functions, many of them with a high number of arguments. It's hard to learn and remember. Frameworks require a new approach to the programming, and therefore have a steep learning curve. Glass had to be easy to use, being almost transparent do the user: that's where it name comes from.

•High performance: if an application requires distributed computing, it's because it has a high computational cost. Glass must be as light as possible.

•Network protocol independent: an abstraction of the underlying network protocol makes Glass protocol independent. You can use TCP, UDP, or even more high level systems such as MPI easily. Just instantiate the proper network class. This is very important in a world where applications have hugely different network requirements.

•Reliable and fault tolerant: any distributed computing library must be reliable and fault tolerant. It's unacceptable that the failure of a node will bring down the entire computational system. Glass had to provide fault tolerance automatically, and be reliable to make sure that faults would be as rare as possible.

•Completely thread safe: many APIs are note thread safe. This forces the user to find workarounds or use other solutions in place of threads. Glass had to work in thread applications seamlessly.

Motivation

- Many libraries and frameworks around
- Why another one?
  - Most are hard to learn or use
  - Few support for graphical applications needs
    - Low delays, synchronization, etc
  - Those that do are too specific
  - Frameworks require complete rewrite of legacy code
  - Even if not so, it's not trivial to port existing code
- Born from DICElib

With many libraries and frameworks out there, why write another one? First, as mentioned, most are hard to learn and use.

Second, one of Glass' goals was running graphical applications on clusters. Graphical applications have specific requirements which include: low delays, fast synchronization, good bandwidth, etc. Not all available APIs fit these requirements. And those that do are usually too specific: they target multiprojection, for instance, not giving much importance to the underlying computational process. They usually provide good resources for graphics (such as scene graphs), but are not good for a intensive physical simulation.

The frameworks around require rewriting the existing code, since it has to adapt to the new paradigm. Most of them consider the application an object, and so the application has to fit a certain API. Legacy code may take a good time to port, and many times the result is a kludge.

Glass was born out of DICElib, a previous project of the Integrable Systems Laboratory of Polytechnic School of the University of São Paulo. DICElib was a small library that solved a few specific problems of multiprojection applications running on clusters; due to its inflexibility and limitations, we decided to write something new. DICElib was an important step, however, to find out the things we really needed and to learn from our mistakes.

Overview

- Written in C++
  - Easy to interface with C, C++, Java, etc
- Glass core provides internal functionality
  - Network system
  - Plug-in management
  - Node management
- User never sees or knows about it

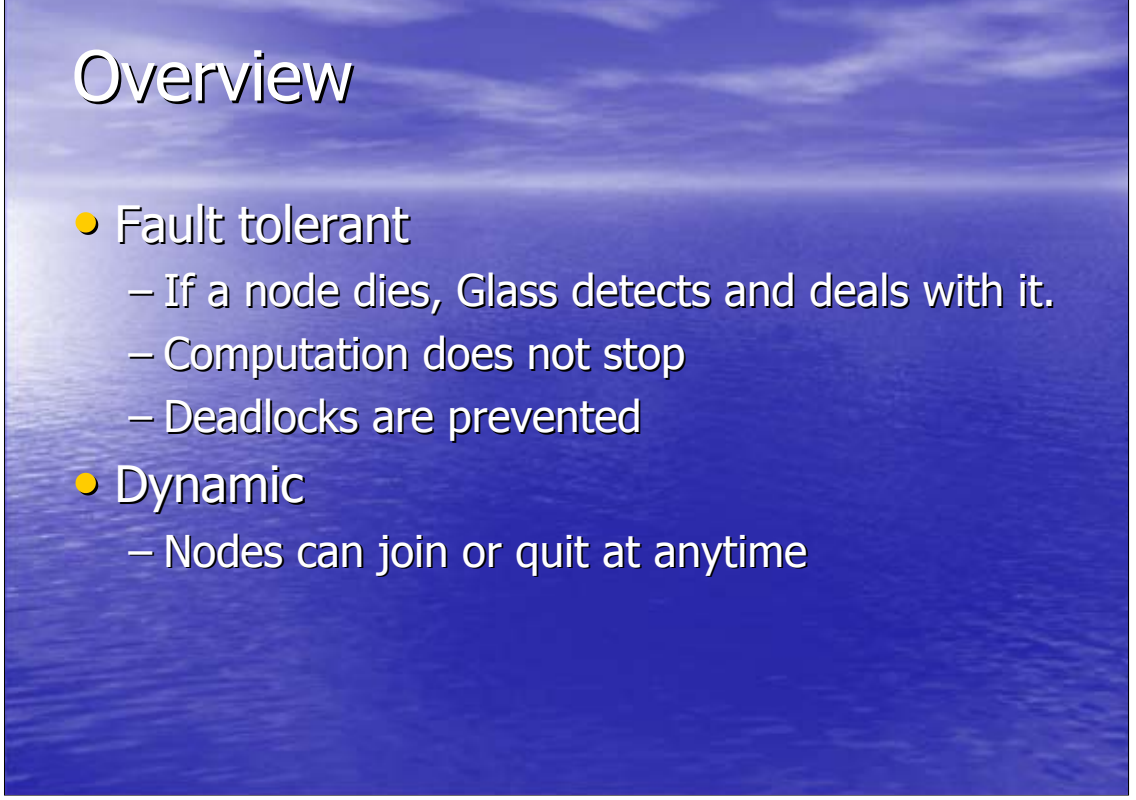
Glass is completely written in C++. The choice was due to it's wide number of compilers and spread use, good performance, and certain features that would make the life of the programmer easier (such as OO and templates). C++ is easy to interface with C, Java and other languages, but C++ is the choice language for most of Glass target current applications.

A core provides all internal functionality, such as the network system, plugin managements (more about plugins in a moment), node management, etc. The user is completely oblivious to this core.

Overview

- A class is responsible for the node behaviour
  - Server
  - Client
  - etc
- Simple, yet flexible
  - Just change this class and you change the network topology
  - No other changes necessary!

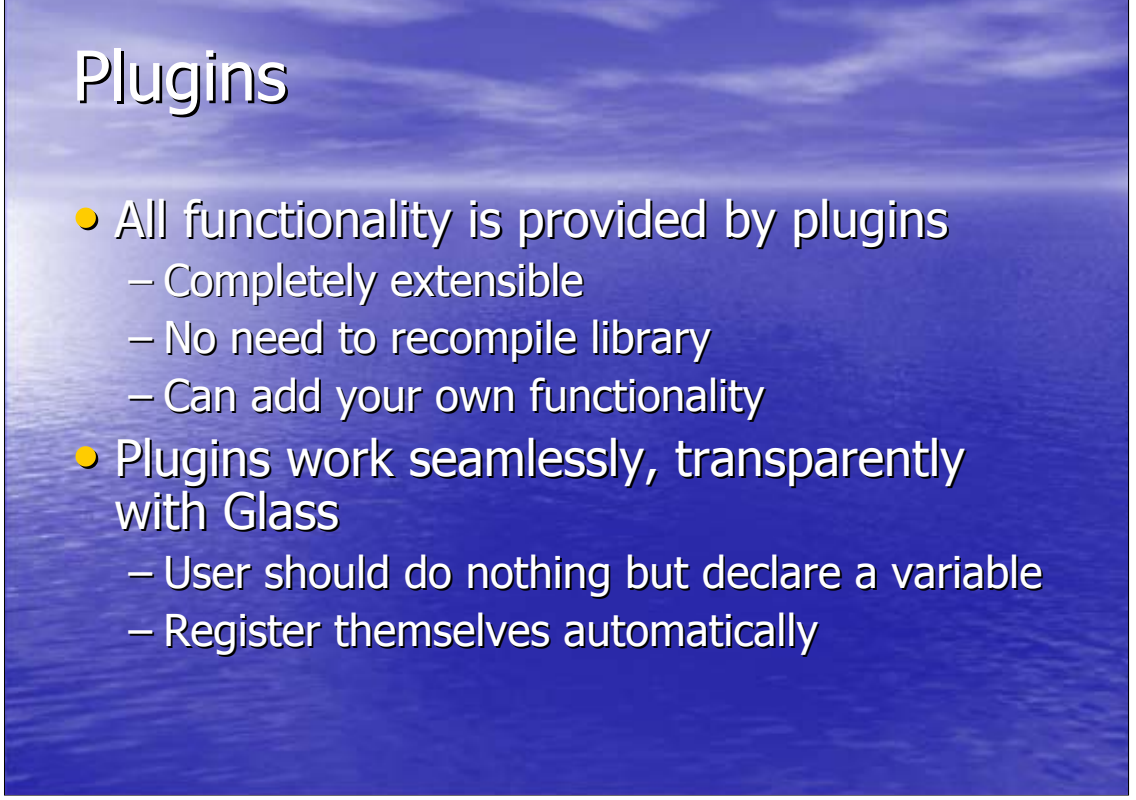
All he does is instantiate a class that is responsible for the behaviour of that particular node. For example, the node may be a server, or a client, or a peer. You can write new classes easily, changing the underlying network topology to fit your needs and gain performance. It's a simple system, but quite effective, and unparalleled.



Fault tolerance is a important feature of Glass. Our idea is that only a catastrophic event such as your network switch blowing up would stop your application.

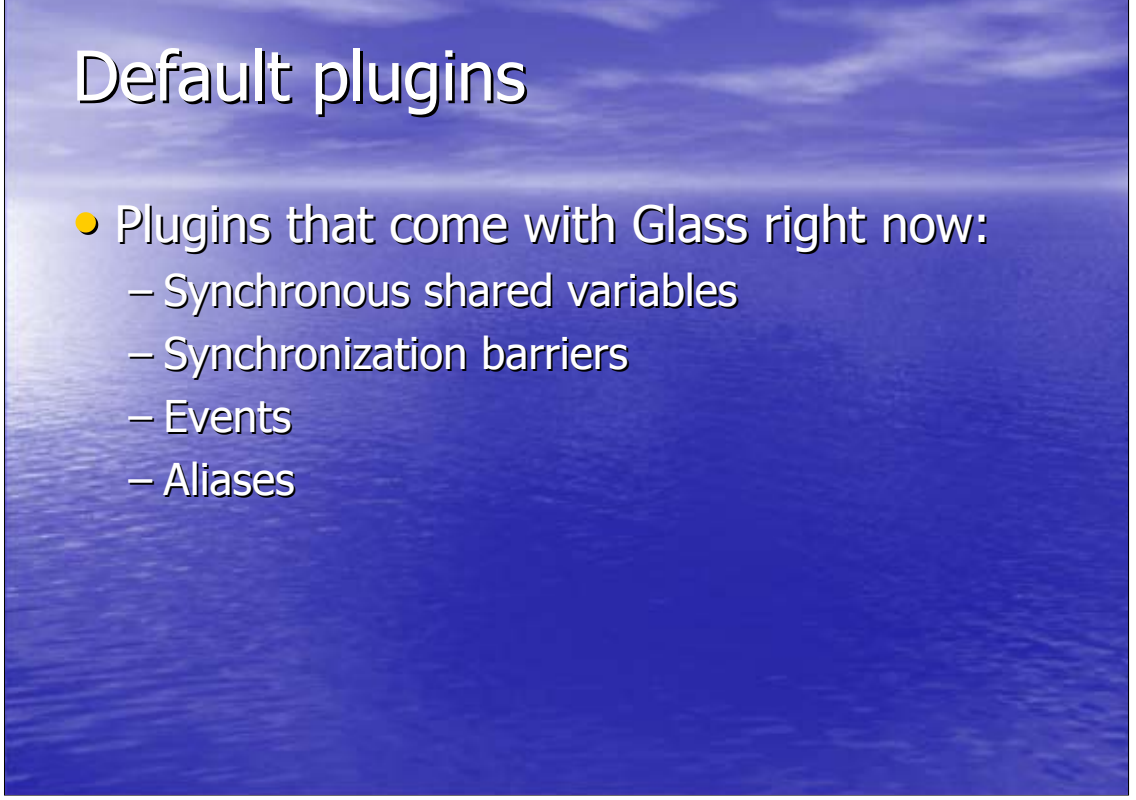
If a node dies, for an example --- for whatever reason --- Glass detects and deals with it. The computation does not stop, and resources held by that node (synchronization barriers, for instance) are automatically freed, avoiding deadlocks. This system is transparent to the user.

Something that is annoying in many APIs is that nodes may only join the computation at a specific time, when starting the application. In a world which is moving to grid systems, this is unacceptable. In Glass, nodes can join or quit at anytime. Deadlocks are prevented automatically. Initial synchronization is made automatically, no matter if your distributed application is a realtime renderer or does a very long offline simulation.



All functionality of Glass is provided by plugins. This is what makes Glass an extensible, flexible system. The plugin system does not require recompilation of the library: they can be part of your executable, or an external library. It's possible, then, to extend Glass to fit your exact needs in a simple, straightforward manner.

The plugins are autonomous, and register themselves automatically. The user doesn't have to know anything about them, initialize them: to him, plugins are transparent and accessed by a simple API, usually looking like a variable declaration. We are going to see some examples.



Glass right now has four tested, stable plugins:

- Synchronous shared variables
- Synchronization barriers
- Events
- Aliases

Shared variables

- Extremely useful for small amounts of data
- Synchronous: avoids consistency problems
- Easy to use
- Type independent
  - Can be of user defined types (classes, structs)

Shared variables are a form of shared memory. They are extremely useful for certain kinds of applications, and are usually efficient for small amounts of data (up to a few Mb), although nothing prevents you from sharing gigabytes. These variables are synchronous: the user specifies when they should be updated globally, and when the variables should be synchronized with the global value. This system is easy to use and avoids all sorts of consistency problems that plague shared memory systems.

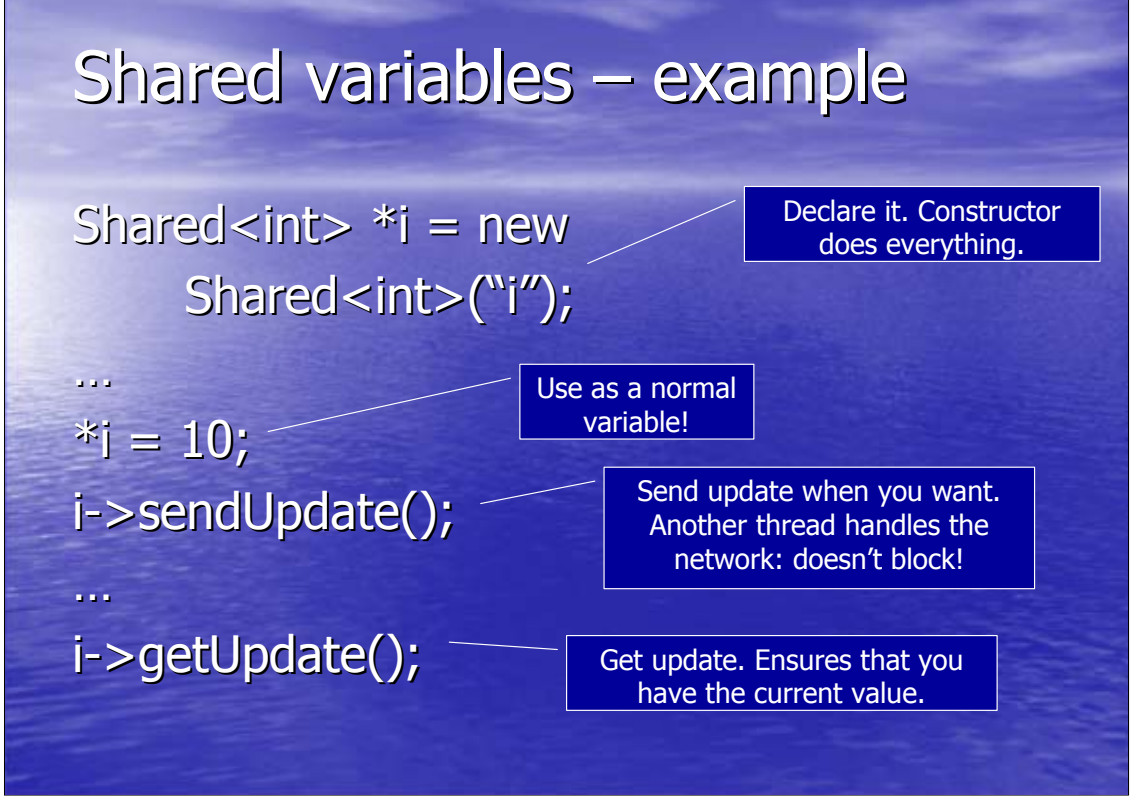
Shared variables are type independent: you can use them with floats, ints, classes, structs, whatever you want. You can provide your own serialization functions. The use of templates and OO features such as operator overloading make the syntax easy.

Shared variables

- Smart
  - Don’t send unneeded data
  - Prefetch/cache data
  - Handle concurrent writing in a predictable way

Shared variables have some smartness built into them. They don’t send unneeded data: only if the variable was changed data is sent. This frees the user of worries of network utilization and optimization. They can prefetch and cache data, effectively reducing delays and balancing network usage.

Concurrent writing is handled in a predictable way: each Glass node has an associated id, which is a positive integer. Smaller integers have priority, so if two nodes write to the same variable, the one with smallest id will prevail. Thus the final value is the same, whatever order the writes were made. This consistent behaviour avoids Heisenbugs.



Here's a sample code of Shared variables. You declare it pretty much as a normal variable --- the only difference is that you enclose the type with Shared<>, and provide a string name, so it can be identified in other nodes. As you can see, you can use it as a normal variable. This makes shared variables just as easy to use as normal variables. To send the current local value, updating the global value, you call sendUpdate. When you want to get the current global value, call getUpdate. Simple and effective. Global consistency can be achieved with the used of a synchronization barrier.

Synchronization barriers

- Important for graphic applications
  - Datalock
  - Framelock
- Can have up to 2^{64} distinct barriers
- Easy way to deadlock
  - Glass does the possible to avoid it
  - System to ensure that even late starters synchronize correctly

Synchronization barriers are crucial for graphic applications, where you have to datalock and framelock (genlock is hardly ever done with software, due to its nature). Glass provides a simple, fast, effective barrier system. Up to 2^64 different barriers can be used, so you can have different synchronization points in your code and make sure that they are never confused.

Synchronization barriers are an easy way to deadlock your program: all it takes is that one node calls a barrier and another call some other barrier. This is a problem specially when not all nodes start to compute together: late starters will most likely deadlock if nothing is done, since there are no guarantees that when they call their first barrier, the other nodes will do that too. Glass, however, treats this situation transparently, ensuring that even late starters synchronize correctly and do not deadlock.

If a deadlock happens, however, Glass can detect it. We are researching the best way to deal with this problem.

Synchronized barriers – example

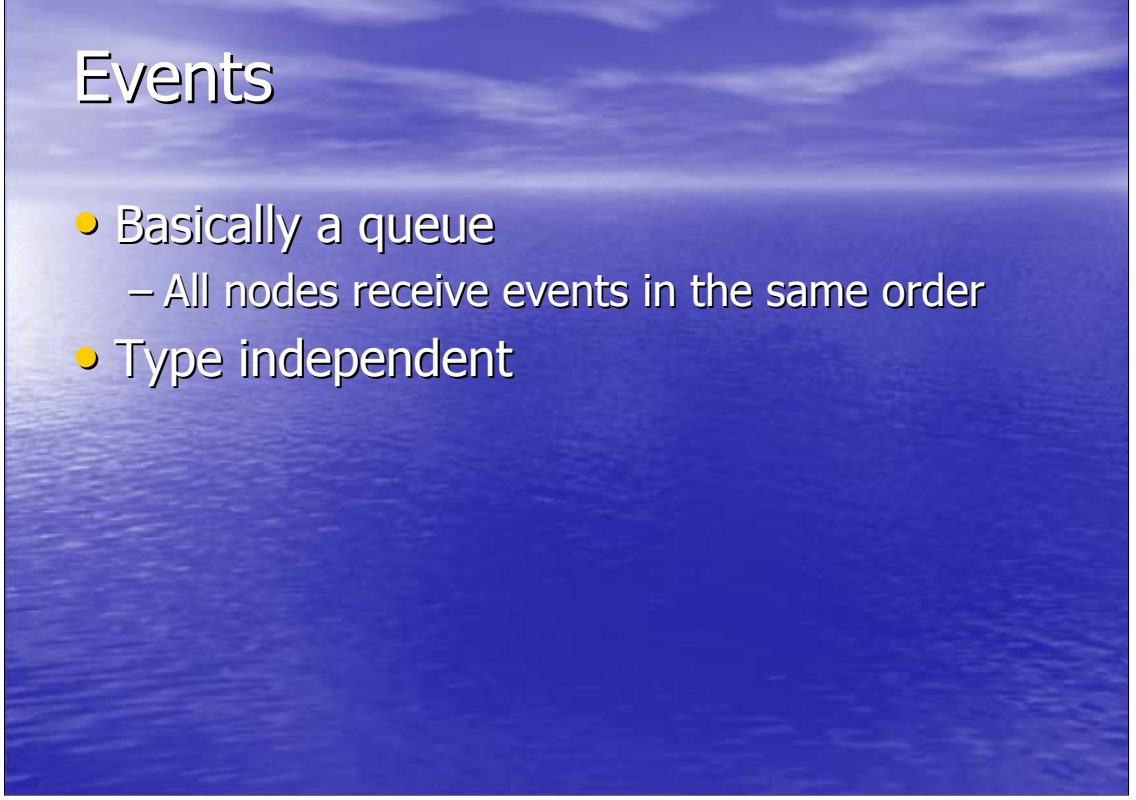
```
Barrier b = new Barrier(1);
Barrier c = new Barrier(2);
...
b->sync();
...
c->sync();
```

Declare barriers. You provide the id here.

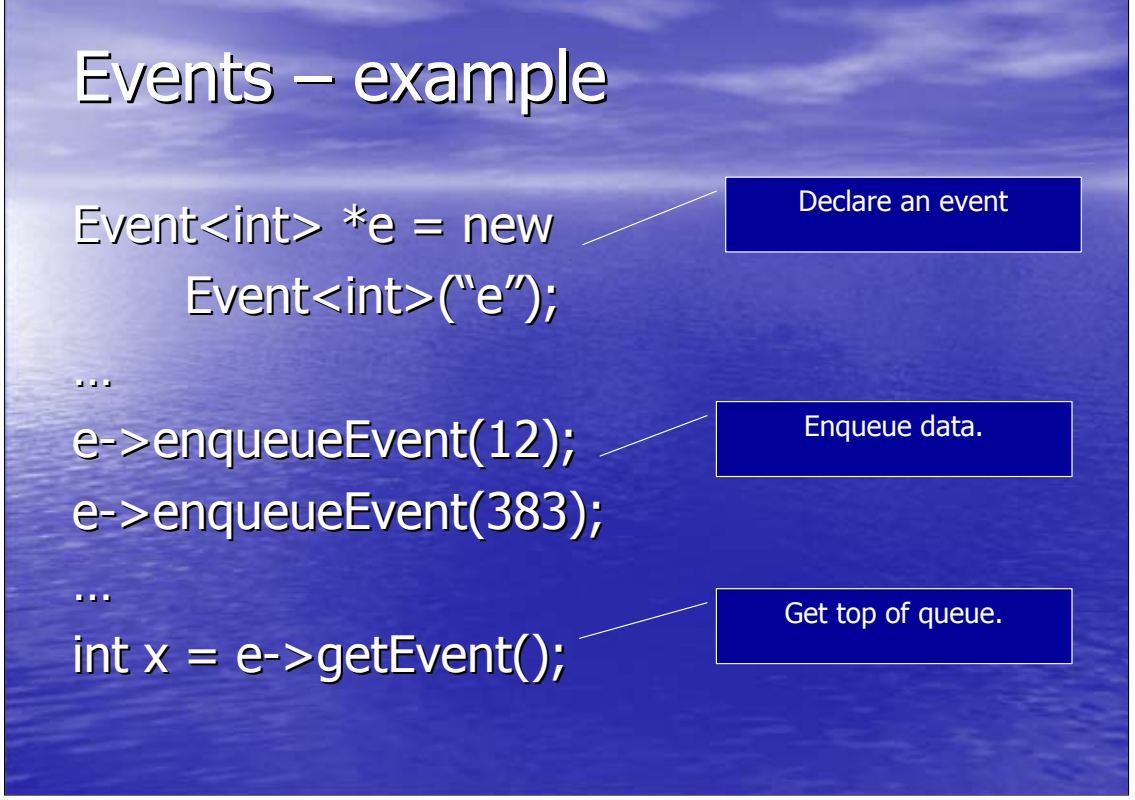
Synchronize 1st barrier.

Synchronize 2nd barrier.

Here's an example of an application that uses two synchronization barriers. They can be, for example, datalock and framelock of a graphical application, called sucessively in an infinite loop. Glass takes care that a new node will correctly synchronize and not deadlock even if, when it starts and calls b->sync(), the other nodes are calling c->sync().



Events are basically a queue, or FIFO, of values. They are quite useful for propagating asynchronous changes or events in an application --- there's plenty of them in interactive applications, like input events (keyboards, mice, etc). Just like shared variables, they are type independent.



It should be quite familiar by now: it's the same syntax used in shared variables. You can enqueue events, which are immediately sent to all nodes that have that event declared. The events are stored in an internal FIFO, and you can get them when you want. Optionally, you can set a callback that is called whenever an event arrives.

Aliases

- Multiprojection environments
  - How to make each node calculate a different view?
- Solution: aliases
  - Same function, different behaviour on each node
  - Works with variables too: same variable, different values
- Original approach
- Similar to pointers
  - But remote, clean, safe
- Great for load balancing!

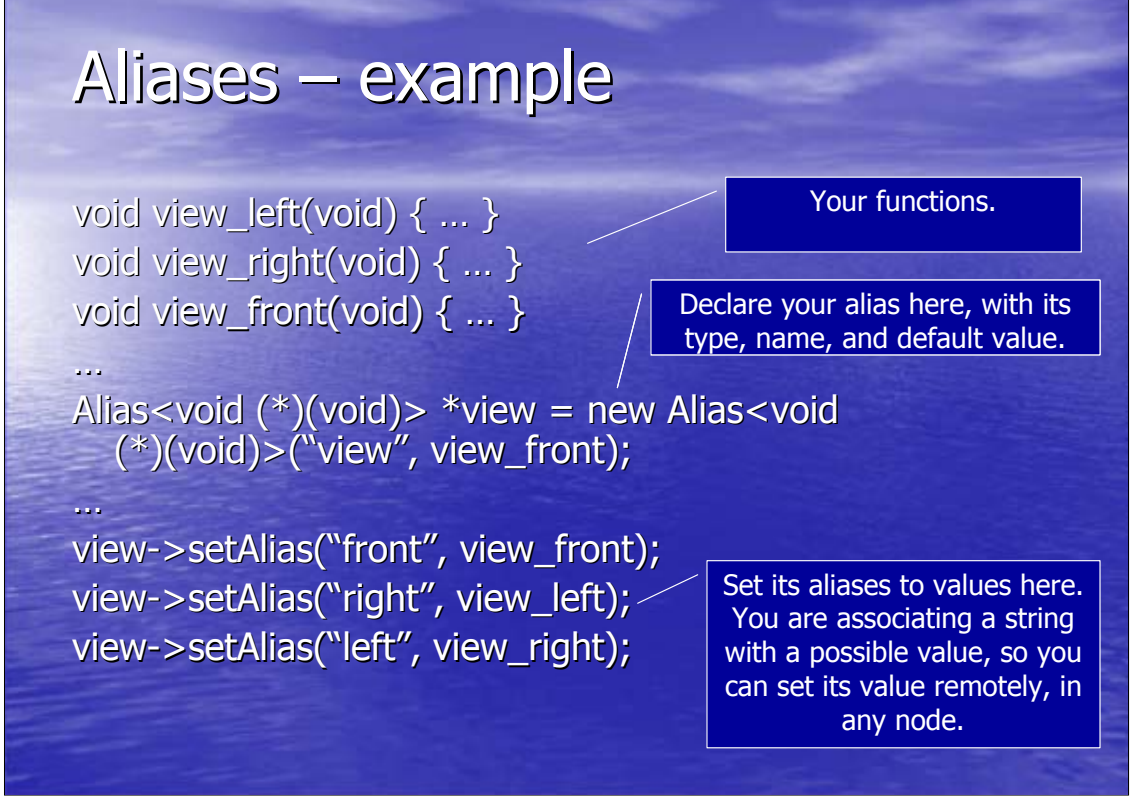
One of the major problems of multiprojection environments is: how do you tell each node to calculate a different view? We wanted a clean, flexible solution, not a hardcoded kludge. Our solution is aliases.

Let's say that I have three different nodes, rendering the front, left and right views, respectively. What if I could just call a function, set_view(), which would automatically find out which view we should calculate? This function would set up the camera, and then you would call your render() function. That's what aliases do. They are a single function, with different behaviour on each node.

Aliases can be variables, too. It's the same idea: the same variable has different values on each node, but you can control these values remotely. This is an original approach: it was first used in a very limited way by DICElib.

You can think of aliases as pointers: you have a pool of possible values, and you choose which one should be used. But, unlike pointers, aliases awork remotely, are clean and safe.

There are many uses for aliases! Since you can change the value of an alias in a remote node, they are great for node balancing.



Here's an example of aliases. Remember that multiprojection application I told you about? Here's its core. You have the three functions that set the view apropriatedly.

Then you instantiate your alias. Call it "view". It's type is void (*)(void), that is, a function that as no arguments or return value. You need to provide a default value: we don't allow NULL aliases here, making them safe. Whenever you call it, Glass guarantees a valid result.

Next step: define the possible values of an alias, associating a keyword with each of them. This provides a way to set values of alias in other nodes, which is rather important when you want to do load balancing, or be able to change the view being rendered on another node.

Aliases – example

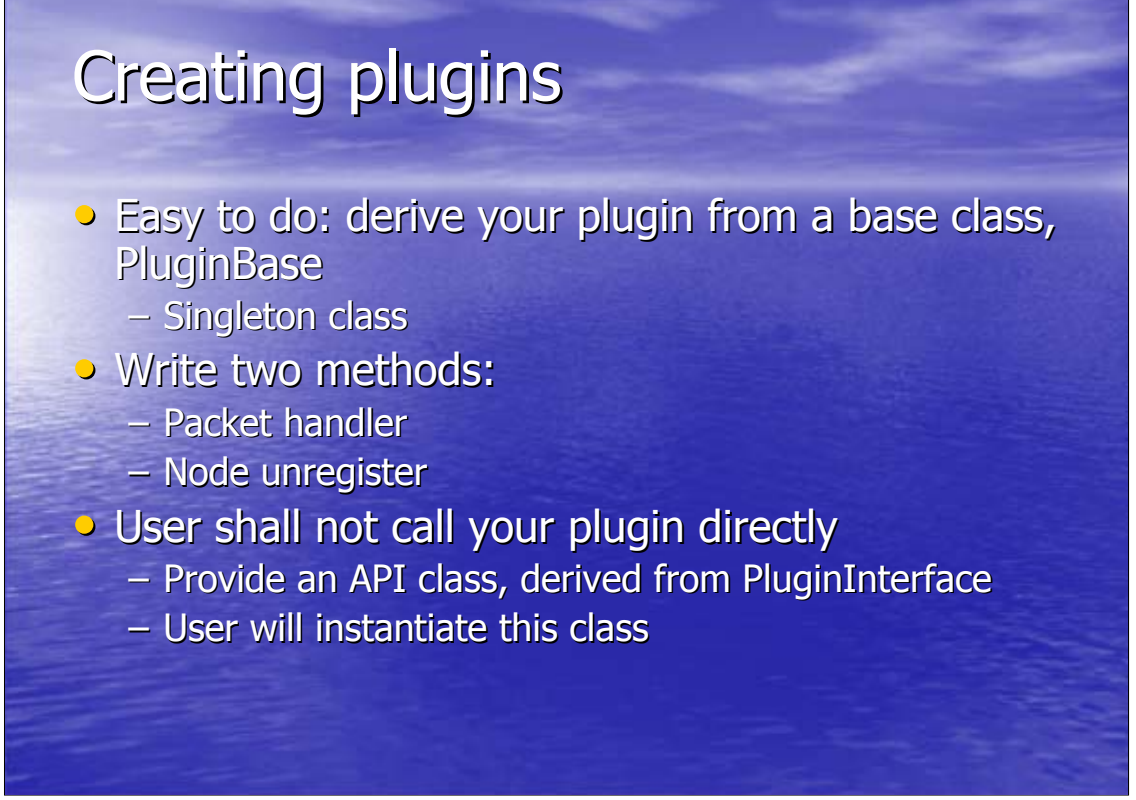
```
view->setValue(3, "front");
view->setValue(4, "right");
view->setValue(5, "left");
...
void (*func)(void) = view->getValue();
func();
```

Set the values of target node. The changes happens immediately.

Get the value of the alias in this node, and then call it to set our view. These two lines should go in your render loop.

Our last step in setting up aliases is actually setting the values on each node. Here we want node 3 rendering the front view, node 4 rendering the right view, and node 5 rendering the left view. You can change these values at any time, and you can set two nodes with the same value.

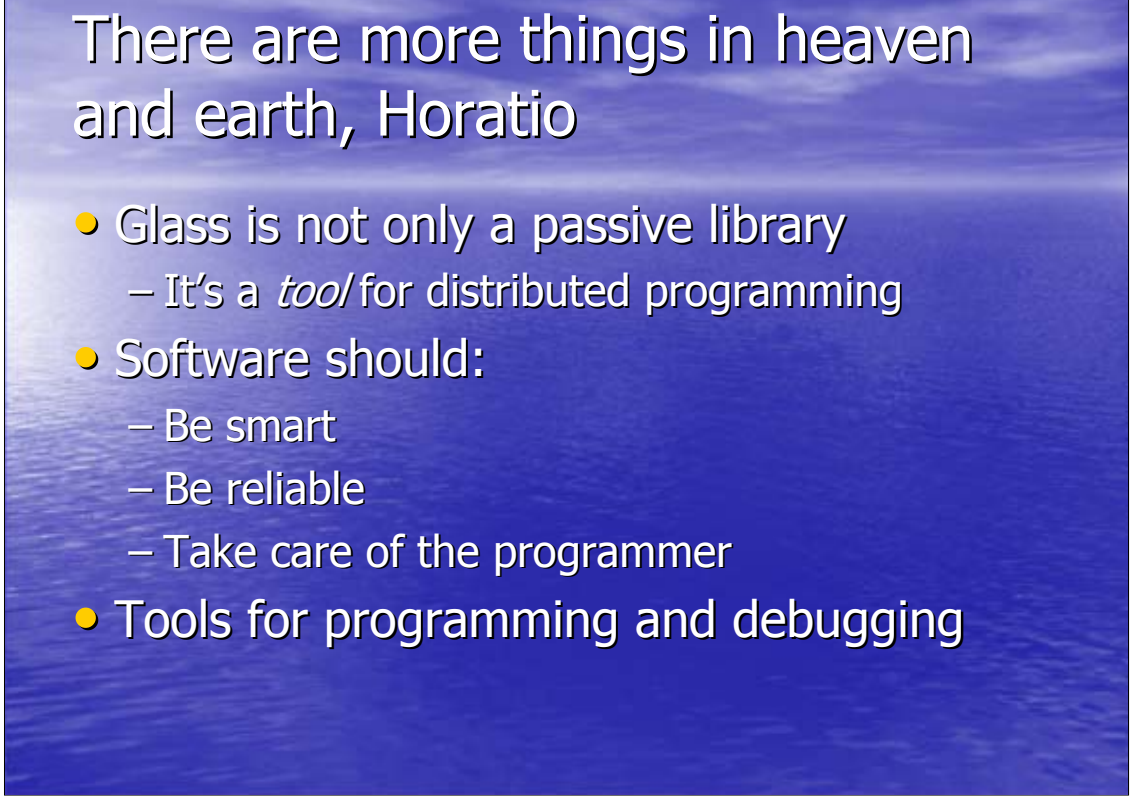
Now that everything is ready, you can use aliases. Just get its current value and use it.



Plugins are easy to create: derive your plugin from a base class, PluginBase. Each plugin has only a single instance running per Glass instance, so it's a singleton. This is because the user should not call the plugin directly, but using an interface class. This design provides a separation between the interface and the functionality, and hides complexity from the user.

A plugin has two virtual methods that you must override: the packet handler, responsible for processing network packets that arrived for this plugin, and a method to unregister any nodes that quit the computation --- this is part of the fault tolerance system; this way, even if a node dies, the plugins guarantee their internal integrity.

The interface base class is a very simple class that provides a few protected methods, invisible to the user. These methods can be used to access data from Glass, from the parent plugin, to send data across the network, etc.



Glass is not a simple library: it's a tool for distributed programming. In other words, Glass has to **help** the user.

In the opinion of authors, software should not be passive. It should:

•Be smart: do things automatically, solve problems autonomously.

•Be reliable: fault tolerance is a must in distributed programming; Glass has been thoroughly tested, guaranteeing a solid core.

•Take care of the programmer: the software should take care of the programmer, and not the opposite. The programmer should use the software as a tool to simplify his job, and not fight with it to achieve what he wants.

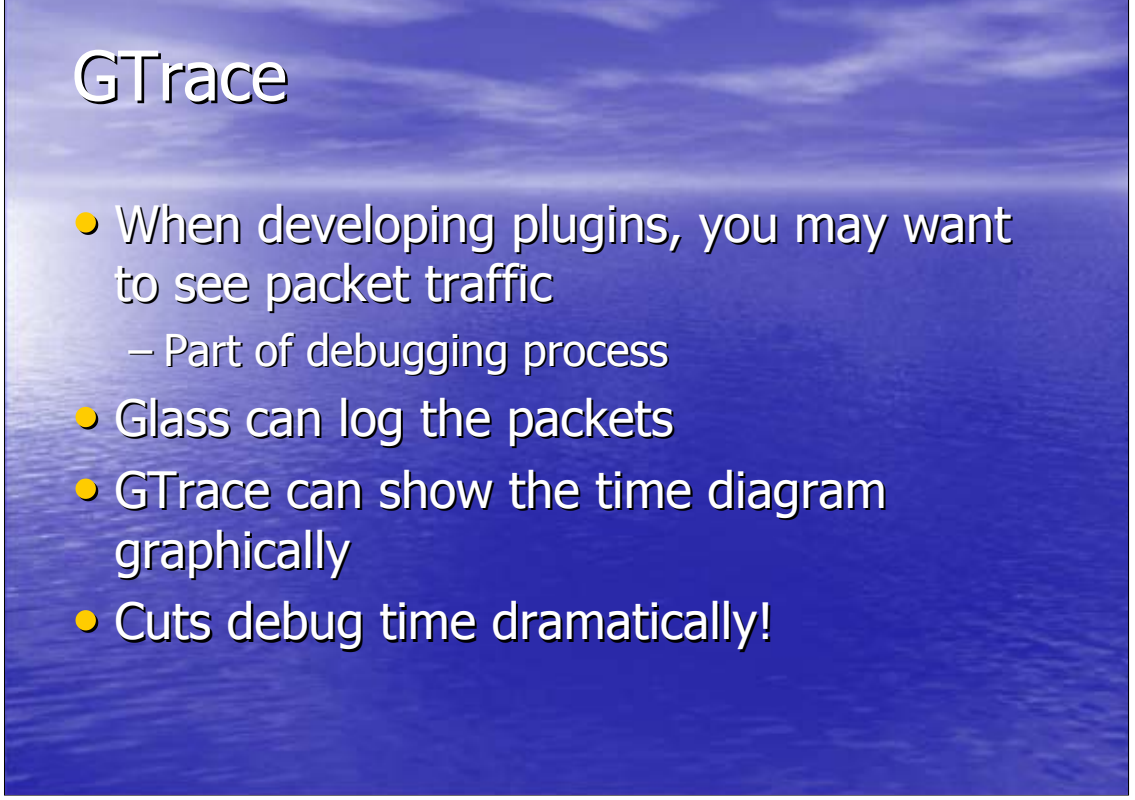
Glass comes with tools for programming and debugging, simplifying its use.

Automatic deadlock detection

- Glass can find deadlocks of its own primitives
- Upon detection:
  - If possible, handle it transparently
    - Such as a node death
  - Otherwise, programs are notified and may restart from a known point
    - This part is still being written

Deadlocking is one of the biggest problems in distributed programming. Even though it's possible to detect deadlocks automatically in most situations, almost no solutions provide this feature. Glass can detect and handle deadlocks automatically, most of the time in a completely transparent way. The programmer doesn't even know that a deadlock occurred or might have occurred: barriers behave this way, for instance, when a node dies or a new node connects.

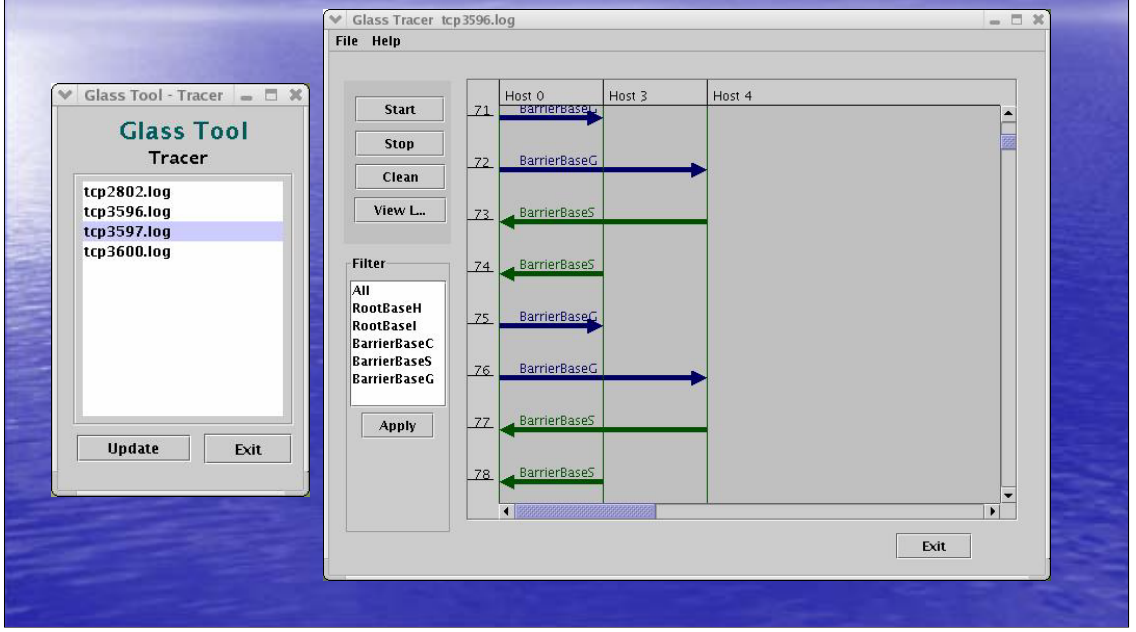
If the deadlock can't be handled in a transparent way, the application is notified and can restart from a known point.



GTrace is a important tool to debug plugins. Your plugins will use the network to send information among nodes, and it's important to see that the packets are being correctly sent when debugging: it's a fast way to find where your code may be wrong.

Glass network abstraction layer can log the packets that are sent and arrive. You can then see the time diagram in a graphical application, called GTrace. It usually takes a glance to find out what packets are wrong: graphical debugging is much easier and faster than reading long text output.

GTrace: screenshot



Here's a screenshot from GTrace: on the left, you can select which nodes you want to see the log. On the left, the packets are shown. Note that you can filter the packets, displaying only those that you are interested in: for example, those of your plugin.

PDA Editor

- Born from desire to control our CAVE with a PDA
- Why not run Glass in it? We do.
- Proof of performance and portability
- Editor generates GUI in a straightforward, graphical way
  - No programming knowledge required

We wanted to control our CAVE applications from a PDA. A PDA is a nice interface to control such applications: it's a very, very rich remote control. The interface is dynamical, easy to use. How to do it? Well, use Glass. Glass' portability and performance makes it possible to run it anywhere: PDAs, embedded devices, desktops, workstations.

We developed a graphical editor, which lets the user generate a GUI to run in the PDA is a straightforward way. No programming knowledge is required to create the GUI!

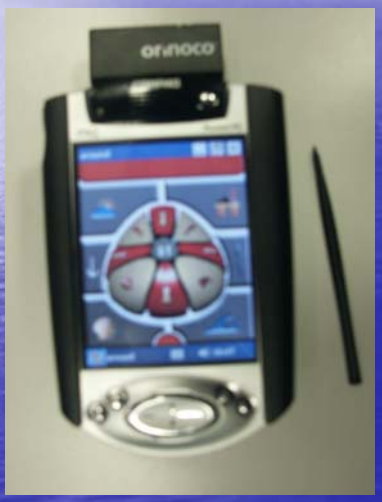
PDA Editor

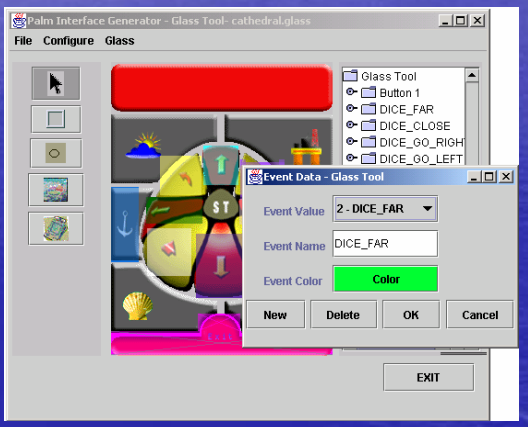
- Code is generated automatically
  - Programmer has only to write an event handler
- Interface is in Java
  - Run it anywhere: PDA or desktop
- Glass is used underneath
  - To Glass, the PDA is just another node

The final code is generated automatically: a skeleton code is generated, which is used on the Glass aplication itself to process the events. It's pretty much like processing keystrokes from a keyboard.

Both the editor and the generated GUI are in Java: it's easy to run it anywhere. Glass is used underneath, via native methods. To the application, the PDA is just another node.

PDA Editor: screenshot

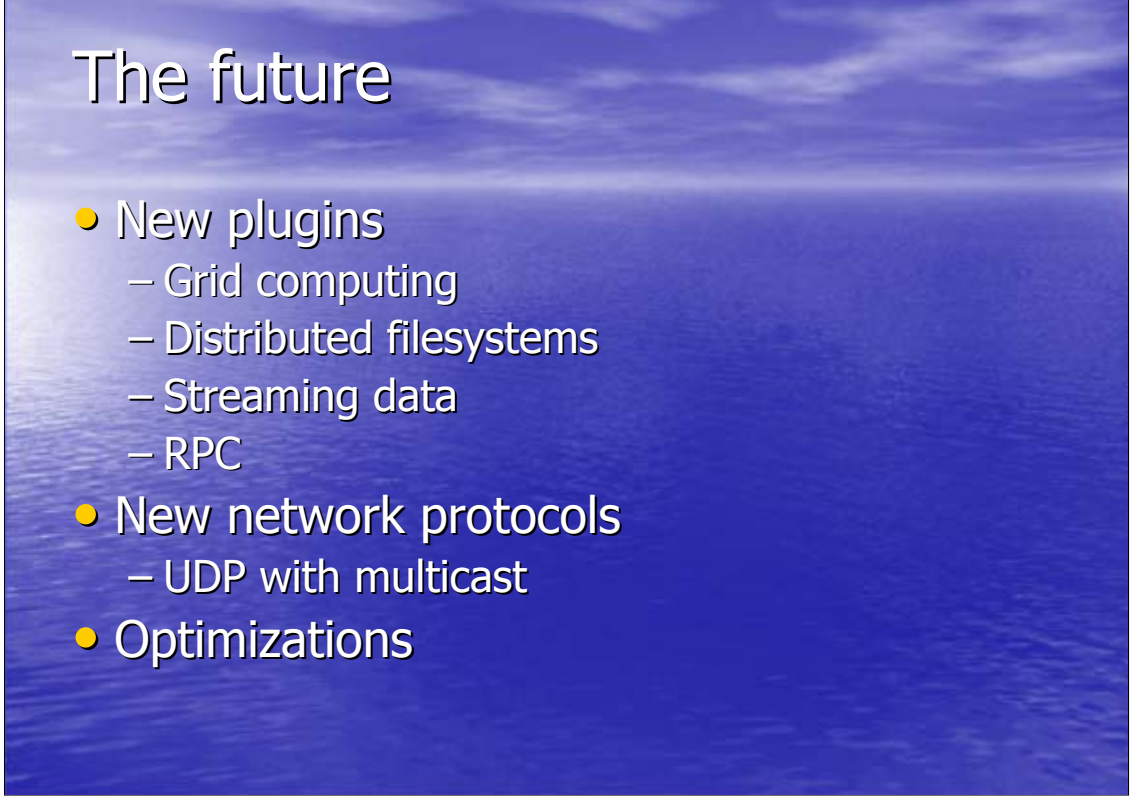




Here are two shots of the PDA system. On the left, a PDA running Glass. On the left is the editor itself. You just have to associate hotspots with events.

What are we doing with Glass right now?

Our main motivation for writing Glass was for running distributed real time graphical applications: virtual reality, scientific visualization, etc. That's what we have been doing lately: the picture shows an architectural visualization system running at our lab's CAVE. Each wall is being rendered by a different node of a PC cluster.



What is in the future?

New plugins are an obvious choice. We plan to add support for grid computing, distributed filesystems, streaming data, RPC and more. New network protocols, such as UDP with multicast or FastTCP are planned. Optimizing the core of the library itself is something else in our to do list.

Conclusions

- Addresses problems that pestered distributed programming for a LONG time
- Programmer should not be bothered
  - Should program only what he REALLY has to
  - API should be high level: it's hard enough to create distributed applications, one should not have to fight with APIs
  - All that can be done automatically must be done so

Glass was designed to solve some problems that have been pestering distributed programming for a long time, and have always been considered a burden that the programmer should carry. In our new paradigm, the programmer should not be bothered. He should only have to program what is REALLY necessary: anything that can be done for him, will be done automatically. The API is high level: there's already enough difficulty in designing and coding distributed algorithms and applications; fighting with an API to do something is counter-productive.

Conclusions

- Porting most RV applications is easy now
  - Simple applications take usually half an hour
- Extensibility is primordial
  - A library should free the user, not constrain him!
  - Whatever you need, you can do with Glass
- Compact, well tested core provides a reliable base

When designing Glass, we were concerned with the existing code base; we saw how difficult was to port some applications to other frameworks. Porting a simple graphical application (say an OpenGL demo) takes about half an hour.

Extensibility is the most important feature, in our opinion: it allows the programmer to modify Glass to fit his exact needs. A library is supposed to free the programmer: when he is constrained by it, something is definitely wrong! Whatever is your need, Glass can do it. Just write a new plugin.

Glass' core is a compact, thoroughly tested code that provides a reliable base.

Conclusions

- Distributed computing MUST be fault tolerant
  - It's unacceptable that failure in one node will halt the entire system
  - This must be done automatically

Last, but not least, distributed computing must be fault tolerant. When you are running an application on tens of nodes, it's unacceptable that a single problem in one node brings the entire application down. The library must be able to handle faults, and, when possible, in a completely automatic and transparent way.

Websites

- DICElib
  - http://www.lsi.usp.br/~brunobg/dicelib/
- Glass
  - http://www.lsi.usp.br/~brunobg/glass/

References

- Guimarães, Marcelo de Paiva; Gnecco, Bruno Barberi; Cabral, Marcio; Bressan, Paulo Alexandre; Zuffo, Marcelo Knörich. **Synchronization and data sharing library for PC clusters.** VR-CLuster'03 - Workshop on Commodity Clusters for Virtual Reality Los Angeles, USA. March 22th-26th.
- Guimarães, Marcelo de Paiva; Gnecco, Bruno Barberi; Zuffo, Marcelo Knörich. **Ferramenta de geração de interfaces gráficas para PDAs.** Simpósio Brasileiro de Realidade Virtual, Ribeirão Preto, Outubro de 2003.
- Gnecco, Bruno Barberi; Guimarães, Marcelo de Paiva; Zuffo, Marcelo Knörich. **Um *framework* flexível e transparente para computação distribuída de alto desempenho**. Simpósio Brasileiro de Realidade Virtual, Ribeirão Preto, Outubro de 2003.