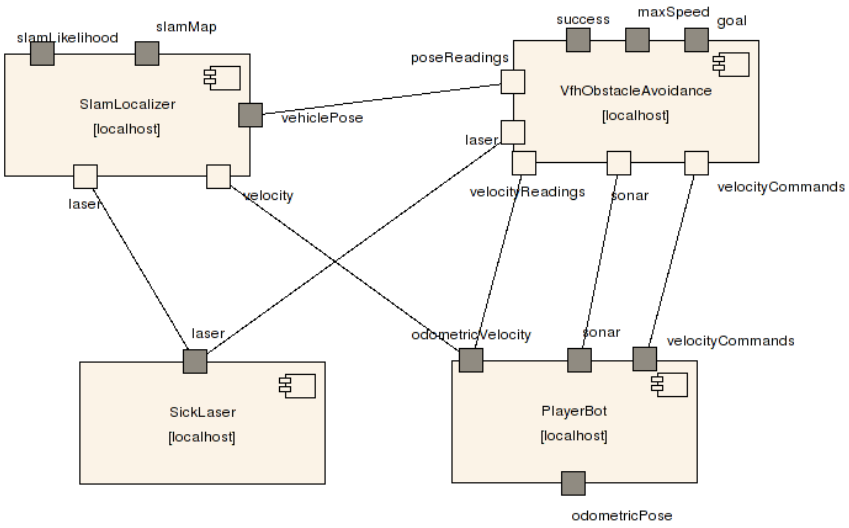
Orca Overview

This page gives a technical overview of Orca. For a non-technical overview detailing information such as Orca's philosophy, history, and licensing, see the About page. For more detailed technical information, see the API documentation on Orca's Objects, Patterns, Components. For the real nitty-gritty, see the Developers' Guide.

Contents

Systems of Interacting Components: An Overview

A component is a stand-alone process. A system consists of a set of components which run asynchronously, passing objects to one another across well-defined interfaces. Components don't know any of the details of any other components, they just provide certain object types and require certain object types. An example Orca system is shown below.



In this image, the small squares by which components are connected are called ports. The only way to communicate with another component is through one of its ports.

Clients and Servers

The words 'client' and 'server' are used frequently, so they are worth defining. The term 'client-server' implies a many-to-one relationship, and nothing more. The terms 'client' and 'server' don't imply anything at all about the direction of information flow.

The 'many' in 'many-to-one' can also be zero. Thus a server can exist with zero clients, but a client needs exactly one server.

Elements of the Framework

The following elements of the Orca framework are important. Care has been taken to ensure that design decisions made about each element do not affect other elements.

1. **Objects**: the abstract definitions of the units of data that are passed (by value) between components. Examples include Point2D, Pose2D and LaserScan.
2. **Communication Patterns**: the abstract policies for how objects are sent between components.
3. **Transport Mechanism**: the method used to physically transport the objects, for example CORBA or raw TCP/IP. Each object's internal representation and each interface type must be implemented at most once for each language and transport mechanism. There is no requirement that all objects and patterns be implemented across all transport mechanisms: an incomplete set simply means that not all components are available for all transport mechanisms.
4. **Components**: The implementations of algorithms and servers for various types of hardware. The components can be implemented with knowledge only of the objects, the object methods, and the communication patterns. The transport mechanism is only relevant at compile-time. The implication of this is that the transport mechanism can be changed at compile-time.

The number of possibilities for each of the items above is virtually infinite. Orca's approach is to define and implement the things which are most likely to be useful, while ensuring that the framework is flexible enough that any of the items above can be added to. Users need only use the parts they like, and are free to make additions where they identify deficiencies.

The four elements are described in further detail below.

Objects

An Object is something which can be passed (through a port) to other components. It knows how to serialize itself ready for transmission to components running on other hosts, and it knows how to put itself back together on the other side (collectively known as marshaling and demarshaling). It knows how to handle endian issues between hosts. Some objects also have useful methods associated with them (such as adding two points).

Since objects are the only common representation shared between components they should be thought out carefully. A proliferation of subtly different objects hampers compatibility between components.

For more detailed information, see the Objects Documentation.

Communication Patterns

Communication patterns are used to define interfaces.

In order for components to be replaceable, their interfaces must be very strictly defined. Orca supports binary compatibility: one component of a system can be exchanged without having to compile anything else in the system.

Components can only be connected across identical interfaces. The definitions of patterns, interfaces and ports might be a little confusing, so the definitions are followed by an example.

Definitions

- **Communication Pattern**: an abstract policy for how objects are sent between components.
- **Interface**: the combination of a communication pattern and an object type.
- **Port**: An entity that implements a specific interface. Implementing an interface may require a choice between additional parameters, such as the port's role in the pattern and how it deals with incoming objects.

Ports are the local entities that components instantiate. Components deal with these ports instead of dealing with remote components directly, and therefore don't have to worry about things like network details.

An Example

Consider the port '`ServerPush_Supplier<Pose2D>`'. This port is used to send objects asynchronously to zero or more clients. Transfer is initiated by the server (the supplier of objects in this case). This port type is made up of the following parts:

```
ServerPush_Supplier<Pose2D>
```

This port name has the following parts:

- **ServerPush**: This is the communication pattern, it defines how objects are sent.
- **Pose2D**: This is the object. This part defines what can be sent through this port.
- The interface is fully-defined by the two items above.
- **Supplier**: This is the port's role in the pattern. The combination of this role and the interface full-define this port.

As another examples, consider the port, '`ClientPush_ConsumerProxy<Pose2D>`'. This port is used to set up an anonymous receiver, to which zero or more clients can send objects. When objects are received the latest one is stored, such that the receiving component can retrieve them at its leisure. This port type is made up of the following parts:

```
ClientPush_ConsumerProxy<Point2D>
```

This port name has the following parts:

- **ClientPush**: This is the communication pattern, it defines how objects are sent.
- **Point2D**: This is the object. This part defines what can be received through this port.
- The interface is fully-defined by the two items above.
- **ConsumerProxy**: This defines both the port's role in the pattern and how it deals with incoming objects. The combination of this role and the interface fully-define this port.

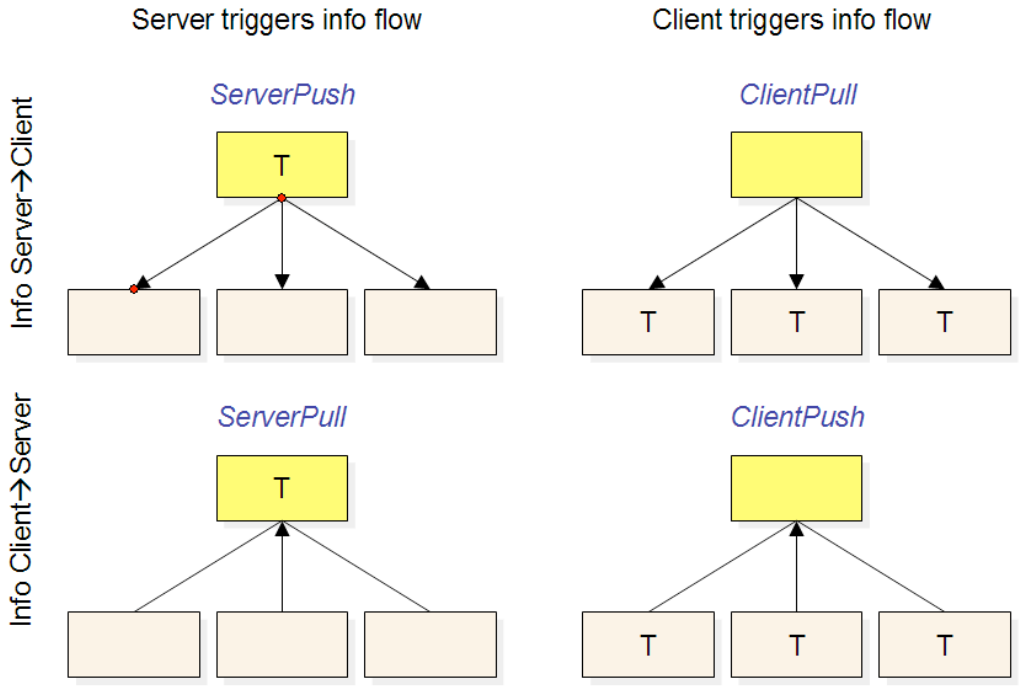
The abstract definitions of the ports reside in `orca-base/src/patterns`. This directory contains a set of abstract classes prefixed with 'I'. Each transport mechanism inherits from these classes and provides their implementations. Templates are used to parameterize patterns over objects to produce interfaces.

For more detailed information, see the Patterns Documentation.

In addition, there is a working example for each of Orca's ports. See these examples for the details of how to instantiate them.

Patterns That Currently Exist

There are many possible way to classify communication patters. The picture below illustrates two axes: a) information flow and b) which sides triggers information transfer. In all four cases the topology is the same: one-to-many with the server on the "one" side and the clients on the "many" side. When the supplier of information triggers the transmission, the pattern is commonly called a "push", otherwise it is called a "pull".



The only patterns that have been implemented currently are ServerPush and ClientPush. In these two patterns, either port can take the role of either supplier or consumer of objects.

For ServerPush, the ServerPushSupplier is the the server, and the ServerPushConsumer is the client

For ClientPush, the ServerPushSupplier is the the client, and the ServerPushConsumer is the server

On the consumer side, the component may desire various types of behaviour when objects arrive:

- **Notify**: The component should be alerted immediately.
- **Proxy**: The incoming object should be stored for retrieval at the component's leisure. When a new object arrive the previous object is over-written.
- **Buffer**: When objects arrive they are stored in a buffer for retrieval at the component's leisure. Since multiple objects can be stored, the component can be sure of not missing any incoming objects.

Currently only Notify and Proxy have been implemented.

Transport Mechanisms

Transport mechanisms provide the means for sending objects, connecting components, etc. There are currently three Orca transport mechanisms: CORBA, CRUD, and SOCKET. Broadly, they have the following features:

- CORBA
  - TCP/IP
    - Anywhere in the world distribution
    - Arbitrary object size
  - Centralized naming service.
  - Always uses the network stack (even on the same host).
  - Significant memory overhead.
- CRUD
  - UDP/IP
    - Limited to a single subnet
    - Object size is limited to the UDP packet size (order of several KB)
  - Built-in decentralized naming service.
  - Uses shared memory for comms between components on the same host.
  - Light-weight
- SOCKET
  - TCP/IP
    - Anywhere in the world distribution
    - Arbitrary object size
  - Centralized naming service.
  - Always uses the network stack (even on the same host).
  - Light-weight

Components

A component is nothing more than a process with a set of ports.

The first thing a component should do when initializing is to read in its XML configuration file:

```
//
// Get xml doc from command line
//
orca::XmlDoc xmlDoc( argc, argv );
```

When the XmlDoc is instantiated, it is automatically validated against the component's schema, reducing the number of checks a component writer needs to do in code.

For the rest of this example, assume that the xml file contains the following information:

```
<client tag="laser">
      <clientNameService nsPlatform="local" nsName="laserServer"/>
</client>

<parameters myParam="0.5"/>
```

A component may want to read some of the parameters from its xml file, like so:

```
double myParamVal;
double defaultValue = 5.0;
xmlDoc.getParam( "myParam", myParamVal, defaultValue );
```

Next, the component probably wants to instantiate some ports, in order to communicate with other components. This is done with code similar to the following:

```
//
// Client to receive pushed data
//
ServerPush_ConsumerProxy client;
if ( client.setup( xmlDoc, "laser" ) != ORCA_SUCCESS )
{
    cerr << "ERROR: Couldn't set up laser client" << endl;
    exit(1);
}

if ( client.connect() != ORCA_SUCCESS )
{
    cerr << "ERROR: Couldn't connect to laser" << endl;
    exit(1);
}
```

This uses information from the XML file to configure the connection's endpoints, as described below. For the example xml snippet above, the Naming service is used.

Aside from this, components are pretty-much free to do whatever they like.

For a complete listing of all Orca's components plus further information, see Components Documentation. For beginners, there are several simple Example Components.

Connecting Components

The topology of a system is configured using components' xml configuration files.

Essentially each component's XML configuration file defines a set of ports and a set of parameters. For each port, enough information must be supplied to allow ports to be connected. The format and content of this information will be different for each transport mechanism. For details, see the documentation on Transport Mechanisms.

For more information about how the xml files are structured, see the section on XML configuration file formats.

In principal there are various ways of specifying components' connections:

- **Static Connections**: Using the transport mechanism's addressing scheme, inter-component connections can be stated explicitly in the components' xml files. This might be done using raw IP addresses, Corba's Naming Service, etc, depending on the transport type. This approach makes sense if there is a small number of components and complete control over the connections is desired. This would be the case, for instance, when putting components together to build a single-vehicle architecture.
- **Centralized Dynamic Connections**: If the exact topology is not as important as ensuring that all components have their requirements met, one could use a centralised trader to connect service providers with service consumers, where provided or required services are configured in the XML files. This might be the case for larger-scale systems, where configuring connections by hand is tedious and/or inefficient.
- **Decentralized Dynamic Connections**: In very large scale sensor networks, a centralized trader is both a bottleneck and a potential point of system-wide failure. In this case components may need the ability to trade services directly.

Currently, the only connection methods are:

- **Naming Service:**

  Each port (not component) registers its provided interfaces (ie servers) with a naming service. When clients require those interface, they look

them up by name to find the physical address.

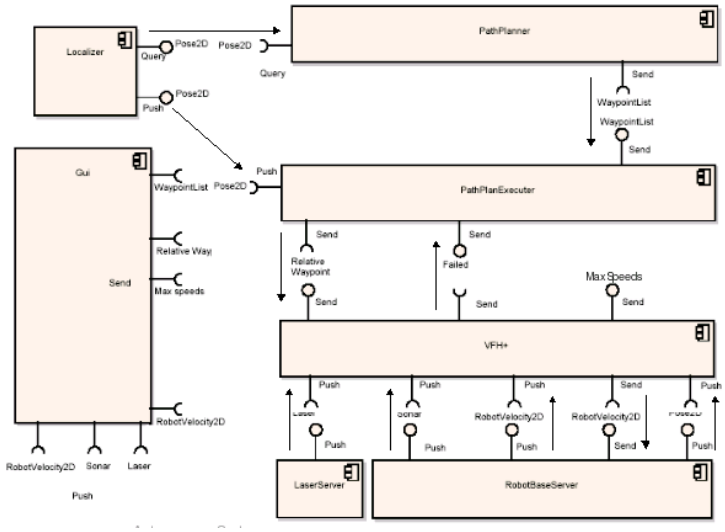
A service's name is made up of two parts: <nsPlatform><nsName>

nsPlatform: Refers to the physical robot. When robotA is run by many computers, setting the nsPlatform of all ports to 'robotA' means that components on robotB needn't know how components are distributed over the computers making up robotA. Since robots are often controlled by a single computer, the special nsPlatform 'local' defaults to the hostname.

nsName: This is the name of the specific service on the robot. The nsNames on a platform must be unique.

Putting It All Together

When you have a set of components who have defined provided and required interfaces you'll want to connect them together into a working system. Since it might not be clear yet what this would look like, here is an example system:



This is a standard three-level architecture for a mobile robot. On the bottom level a VFH+ component implements reactive obstacle avoidance. On the top level a PathPlanner generates a list of waypoints in non-deterministic, unbounded time. A GUI connects at any level of the hierarchy. A localizer is responsible for keeping track of the robot's position.

IMPORTANT: This is just one possibility (one of an infinite set) of Orca-based architectures for a single mobile robot. It is included merely to help solidify the idea of piecing components together into a system.

For practical information about putting a system together, see the step-by-step tutorials.

Units and Senses

Units

All component configuration is done in S.I. units (this means metres, seconds, etc). The only exception to this is angular quantities, which are configured in degrees rather than radians, because humans generally think better in degrees.

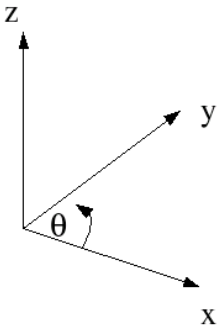
Internally, unless there is a good reason for doing otherwise, S.I. units are used everywhere (including for angular quantities).

Senses

All angles are defined from -pi to pi, NOT 0 to 2pi.

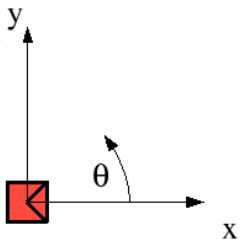
There are two kinds of coordinate frames: frames that are rooted in a global map, and a platform's local coordinate frame.

For worlds where the curvature of the earth is not important, the global coordinate axes are defined as follows:



Angles are defined anti-clockwise from the x-axis, with zero degrees along the x-axis. The y-axis is at +90 degrees. The z-axis points up.

A local frame is defined as follows:



A robot or sensor looks down the x-axis, with the y-axis out to its left. The z-axis points up. Zero degrees is directly in front, with angles increasing to the left and decreasing to the right.

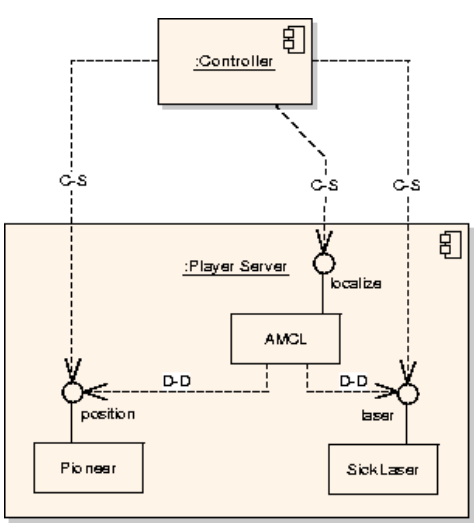
Webmaster: Tobias Kaupp (tobasco at users.sourceforge.net)

Orca FAQ

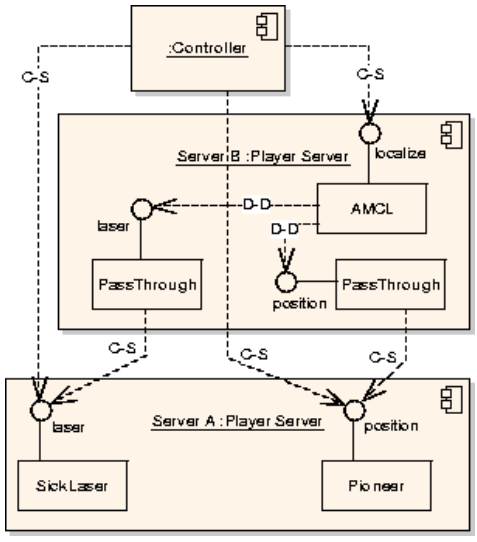
Q:How is Orca similar and/or different from Player?
A: Player is a very successful open-source robotic device server with a large developer and user base. We've used Player (and the accompanying simulator Stage) extensively and know it well, both the strong points and limitations. We believe that Player success can be attributed to the following:

- Simplicity (especially on the client side)
- Broad hardware support
- Distributed hardware access
- A good simulator

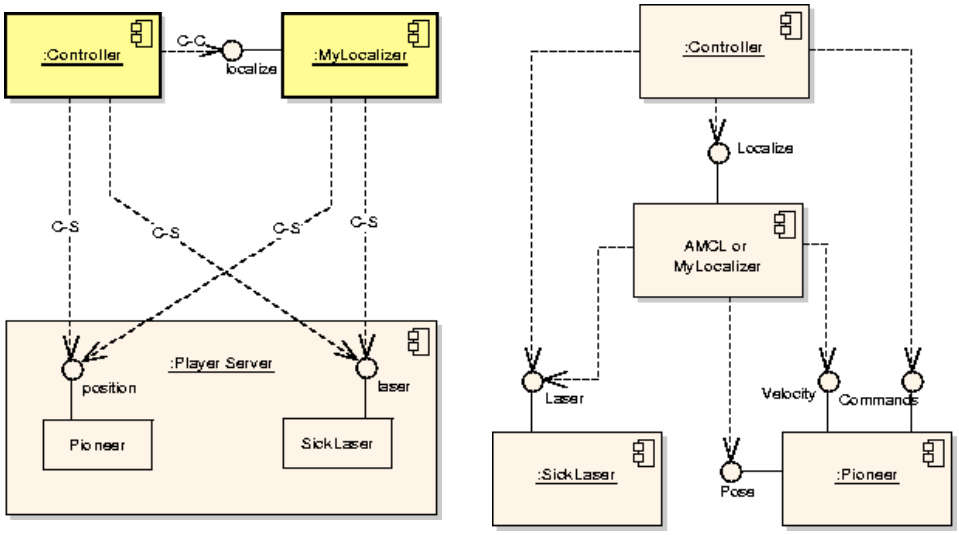
It also has a few limitations. The most important is that there is a fundamental difference between the server space and the client space. Figure 1 illustrates the consequences of this design decision.



(a) The setup using standard Player devices plus a controller in client space.



(b) Moving AMCL to a different host requires extra PassThrough devices.



(c) Using one's own localizer in client space requires the use of client-client communication.

(d) The Orca method: only one type of communication is required.

Fig. 1. Various configurations for a system with hardware servers, a localizer (either Player's Adaptive Monte Carlo Localizer (AMCL) or a hand-written localizer) and a central controller, using either (a)-(c) Player or (d) Orca. The map (required for AMCL) is omitted for clarity. For Player, communication links are labelled Client-Server (C-S), Device-Device (D-D) or Client-Client (C-C). The use of non-standard Client-Client communication in (c) affects the re-useability of both the `MyLocalizer' and `Controller' implementations.

There are several types of communication in the Player diagrams in Figure 1:

1. Inter-device communication, either within a Player server or between servers via a passthrough device
2. Client-Server communication
3. Inter-client communication

Inter-client communication is not addressed by Player at all, and must be invented by users. As such, software modules that rely on ad-hoc inter-client communication mechanisms do not conform to any particular standards, and are not re-useable in the same way as Player devices are. To make the `MyLocalizer' and `Controller' modules independently re-useable under the Player model, MyLocalizer must be ported to server space. This requires significant effort, a possible re-design to fit the Player Device model, and code changes to the Controller.

Rather than developing in client space then porting to server space, one might develop directly in server space. In practice this doesn't happen due to the extra complexity of conforming to the Player device model, and the problems of developing within a monolithic server where where all components must be started and stopped (or seg fault-ed!) together.

The Player model works best when its assumptions, namely that there are server-only modules and client-only modules, are correct. In contrast, all Orca components are equal. Components that act both as servers and clients are much more natural in this model. Since there is only one inter-component communication mechanism, the localizer in Figure 1 can be replaced or moved to another host easily, without requiring changes to other parts of the system.

The consequence of this design decision is that all Orca components are equally re-useable. The differences between Orca and Player become more apparent as system complexity scales from the simple vehicle in Figure 1 to complex distributed systems.

Orca will try to emulate Player's success by copying Player's strong points while trying to remove some of the limitations. Here are the areas where we will follow the Player model:

- Simplicity: the ease of creating new clients in Player is a "gold standard" we will try to achieve by wrapping and hiding the complexity of the underlying transport mechanism. We'll probably not match Player on the client side but we will try to come close. On the server side we hope to do better than Player.

- Hardware: we can gain instant access to all Player-supported hardware by wrapping Player drivers inside Orca components.
- Distribution: UDP and TCP communication will give us the same distribution potential as Player but with finer granularity (individual components not the entire server).
- Simulation: wrapping Player clients will allow us to continue using Stage and Gazebo (the 2D and 3D simulators respectively).

Here're the areas where we can do better than Player:

- Architectures. Every component can be a server, a client or a peer in relation to other components. This is the most important and far-reaching advantage. It is clearly a very useful feature in applications where Player is currently used most: single vehicle architectures. We believe it to be invaluable in widely distributed applications like sensor networks.
- Software Reuse. All components are the same and equally reusable. Because each component runs in its own process, it is easier to experiment with development quality software without endangering the rest of the system, i.e. on a running robot.
- Transport mechanisms. In addition to implementing a light-weight UDP-based transport mechanism, we will try to take advantage of standard distributed middleware. CORBA basic services free the developer from the tedium of writing marshaling/de-marshaling code. It also safeguards against hard-to-track bugs related to cross-platform and cross-language byte-level incompatibilities between data types. The use of CORBA standard Naming and Trading services allows greater flexibility in assembling large systems/architectures without any extra development effort.

Q: How do I create a new object type?
A: It's easy! Look at the webpage describing the Orca objects

---

Q: How do I create a new component?
A: It's also easy! Look at the webpage describing the Orca components

---

Q:I heard that CORBA's really complex to use. Is this true?
A: CORBA is an industrial, heavy-duty tool (see this link for some example TAO users). This has the benefit of being well-tested and therefore robust, and broad enough in scope to handle pretty much any scenario. The downside of this is that it is fairly complex. Orca specializes CORBA from handling pretty much any scenario to handling communications for mobile robotics, hiding almost all the complexity in the process.

Q:I heard that CORBA requires a lot of memory overhead. How much?
A: The size of the TAO orb itself is 2876kb, of which all but 460kb is shared (ref: this link). This means that if a given processor is running one orca component it pays a memory price for TAO of about 3.3Mb, but each additional orca component only pays 460kb. On top of this is orca overhead (much of which is shared) plus the cost of whatever useful code you want.
That said, if you compile ACE+TAO and orca straight out of the box and run the example orca architecture, you get memory footprints of 13Mb for each orca component (about 11.5Mb of which is shared) plus about 7Mb for the NamingService. Tips for slimming TAO down to the sorts of size mentioned above are available in the TAO faq, here.
TAO is one of the heavier open-source ORBs available, however it is the only one (at the time of writing) that contains support for value-types, which orca requires.

Q: What sort of latency can I expect when making calls between orca components?
A: The ORBit people did some comparisons here, which clock 10,000 simple operation invocations (including marshalling/de-marshalling etc) at 8.81 seconds.
Depending on what you need, you don't always need to wait for this however. If you set up a Push server then your client's ServerPush_SupplierProxy will get updated behind the scenes, such that when you need to access its data you'll only be accessing a local copy.

Q:When communicating between components on the same machine, can I use shared memory instead of TCP/IP?
A: There is support in TAO for something called Pluggable Protocols, which addresses this. There is currently no support for this in orca however.

Q:Troubleshooting naming service.
A: If you have problems with the CORBA naming service, it may be that you do not have multicast working. Read the file $TAO_ROOT/orbsvcs/Naming_Service/README, especially "Troubleshooting" near the end. Note that on a host with no network connections you must enable multicast on theloopback interface. The error message looks like this.

```
$ $TAO_ROOT/orbsvcs/Naming_Service/Naming_Service -d -m1
Notifying ImR of startup

We'll become a NameService
subscribe: No such device
Failed to start the Naming Service.
Notifing IMR of Shutdown
```

- Look at the routing table, and enable multicast (note root access).

```
$ /sbin/route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
# /sbin/route add -net 224.0.0.0 netmask 240.0.0.0 dev lo
$ /sbin/route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
224.0.0.0       *               240.0.0.0       U     0      0        0 lo
```

Q:I want to learn more about CORBA - what's a good book?
A: The following are good:

- Henning,Vinoski: Advanced CORBA Programming with C++. This is the classic textbook on corba. Very good but does not deal with any of the newer stuff in recent versions of the corba spec. Highly recommended never the less.
- Bolton: PURE CORBA. This is a good book that also deals with newer stuff.
- Sams' Teach Yourself CORBA in 14 DAYS. This is a good introduction when you are an absolute beginner. Pretty soon you will need another more indepth book though.
- Siegel: Quick CORBA 3. This only handles the newer corba stuff. But it probably is the best one on that.

Webmaster: Tobias Kaupp (tobasco at users.sourceforge.net)