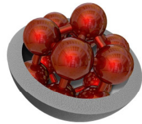


GizmoDistribution

Programmers Manual for the GizmoDistribution Toolkit

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

CONTENTS

1 GENERAL....4
  1.1 What is GizmoDistribution?....4
  1.2 Rationale....4
  1.3 Readers guide....5

2 GIZMODISTRIBUTION OVERVIEW....6
  2.1 Publish-Subscribe networking....6
  2.2 GizmoDistribution fundamentals....7
    2.2.1 Clients....7
    2.2.2 Sessions....7
    2.2.3 Objects....7
    2.2.4 Events....7
    2.2.5 Attributes....7
    2.2.6 Distribution mechanism....7
  2.3 GizmoDistribution components....10
    2.3.1 Libraries....10
    2.3.2 Tools....10

3 GIZMODISTRIBUTION DETAILS....11
  3.1 Classes....11
    3.1.1 gzDistribution classes....11
    3.1.2 gzRemoteDistribution classes....13
  3.2 Distribution manager....14
    3.2.1 The kernel....14
    3.2.2 Starting and stopping....14
    3.2.3 Other tasks....15
  3.3 Sessions....16
    3.3.1 Distributed data-space....16
    3.3.2 Client operations....16
    3.3.3 Sessions types....16
  3.4 Objects....18
    3.4.1 Data stored on sessions....18
    3.4.2 Transactions....19
    3.4.3 Custom types....19
  3.5 Events....20
    3.5.1 Temporary data....20
    3.5.2 Custom types....20
  3.6 Clients....21
    3.6.1 Data user....21
    3.6.2 Initailization....21
    3.6.3 Subscriptions and notifications....22
    3.6.4 Ownership....23
    3.6.5 Threading....23
  3.7 Evaluators....24
  3.8 Servers....25
    3.8.1 States....26
    3.8.2 Priorities....26
  3.9 Remote Distribution....27
    3.9.1 Sessions and servers (revisited)....27
    3.9.2 Channels....28
    3.9.3 Transfer modes....29
    3.9.4 Transports....29
    3.9.5 Custom encoding....30
    3.9.6 Routing / multiplexing....31
  3.10 Reference management....34
  3.11 Iterators....35

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.12 Error management 35
3.13 Debugging 36
3.13.1 gzDistMonitor 36
3.13.2 gzDistSniffer 37

4 TUTORIALS 38
4.1 How to create GizmoDistribution applications 38
4.1.1 The GizmoChat application 38
4.2 How to setup GizmoDistribution projects 45
4.2.1 Microsoft Visual Studio 6 45

5 FAQ 51
5.1 Object and event issues 51
5.2 Client issues 51
5.3 Network issues 52
5.4 General issues 53

6 LINKS 54

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

1 GENERAL

1.1 What is GizmoDistribution?

GizmoDistribution is a software development toolkit for creating distributed software systems in C++. It has emerged from the demands for efficient and reliable real-time data network exchange in the military simulation and training systems developed at SAAB.

GizmoDistribution offers an efficient, reliable and easy to use platform for systems and applications that need real-time network distribution of data. It makes it possible to distribute data between different threads, processes, hosts and networks in a uniform manner. The distribution mechanism implements a data-centric publish/subscribe architecture that decouples publishers from subscribers to enhance flexibility and scalability. It distributes events and object states to make efficient use of available computer resources.

GizmoDistribution is one of the components of the GizmoSDK. GizmoSDK is a cross-platform software development kit for C++ based applications.

1.2 Rationale

- The purpose of GizmoDistribution is to provide programmers with a development kit for creating *distributed software systems in C++*. Data can be shared between clients in a uniform manner no matter where they are located; in a single process, in different processes on the local host or on different hosts on a network.

- The major design goal is to create an *efficient* and *reliable* but yet *easy to use* platform for real-time network distribution of data. Target applications are e.g. distributed simulation, command and control (C4I), distributed control, sensor networks, network gaming, etc.

- GizmoDistribution distributes *events and object states* rather than providing a general-purpose remote procedure call (RPC) based mechanism. The purpose is to keep the design simple and to make efficient use of available resources, e.g. network bandwidth.

- The interfaces of objects and events are defined dynamically in run-time, i.e. no IDL descriptions or other compile time definitions are needed. The purpose is to enhance system flexibility (e.g. backward compatibility), and to make life easier for developers.

- GizmoDistribution implements a *data-centric publish/subscribe* architecture. Data-centric communication decouples publishers and subscribers, i.e. they have no need or interest in knowing who or how many the other publishers and subscribers are and where they are located. The purpose is to enhance *scalability* and *flexibility* of the target systems.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

- GizmoDistribution can be used for simple single user applications as well as complex distributed applications including multiple users and workstations. It is easy to enable distribution to an existing single user application developed with GizmoDistribution. Clients can easily be added, removed or relocated without affecting existing clients.

- Nothing prevents GizmoDistribution from being combined with other similar distributed object solutions within a system. For example both HLA and GizmoDistribution can be used in an application. RPC-based solutions like COM and CORBA are also possible to combine with GizmoDistribution.

- GizmoDistribution is one of the components of the GizmoSDK. It is a *cross-platform* development kit. Write once and run almost everywhere is the philosophy. The components are verified on all recommended platforms to make sure they behave in the same way with memory management, threading, synchronization, networking, etc. that normally is dependant on the selected platform.

1.3 Readers guide

This manual is mainly aimed for software architects, developers and programmers and other potential users of GizmoDistribution.

Chapter 2 gives an overview of the API. It describes fundamental concepts that are important to understand before proceeding with chapter 3 that describes the details of the API. Chapter 4 contains step-by-step tutorials and chapter 5 is an FAQ where you may find answers your questions.

Developers that need a quick introduction on how to use GizmoDistribution should first read chapter 2 to gain an understanding of the concepts. Then it would be a good idea to jump to the tutorials in chapter 4 to learn the basics of how to create distributed applications.

Since GizmoDistribution uses many classes and features from the GizmoBase API, it is recommended to also read the Programmers Manual for GizmoBase. It would also a good idea to have the online reference manuals at hand while reading, since the document includes code snippets illustrating how to use the API.

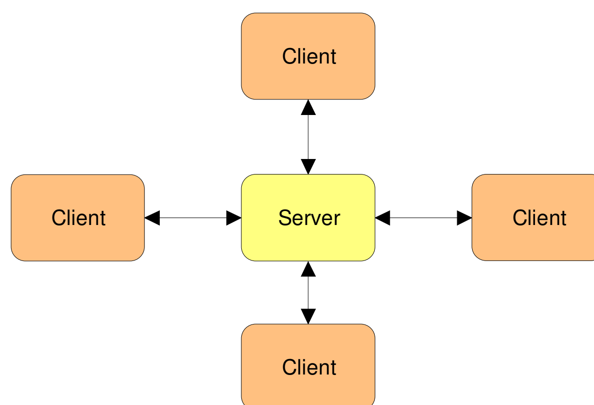
GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

2 GIZMODISTRIBUTION OVERVIEW

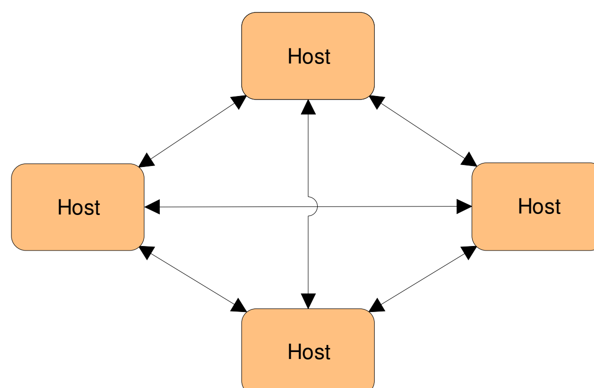
This chapter gives a brief introduction to GizmoDistribution. First the publish-subscribe network distribution paradigm is explained briefly. Then the fundamental concepts of the toolkit are introduced. These concepts are very important to understand before reading any proceeding chapters. Finally the components (libraries and tools) of the toolkit are described briefly.

2.1 Publish-Subscribe networking

There are two basic networking paradigms in common use today; the *client-server* paradigm and the *publish-subscribe* paradigm. In client-server networks are clients connected to a dedicated central server, as shown below. Clients issue requests to the server. The server processes the requests and returns the result to the clients. The client-server paradigm is used by e.g. DCOM and CORBA.



In publish-subscribe networks is there no dedicated central server. Instead data may be distributed from any host to every host in a many-to-many communication pattern as shown below. GizmoDistribution basically use the publish-subscribe networking paradigm, since there is no dedicated server. But to keep objects consistent between processes there must always be (at least) one process that act as a server or master, as described in section 3.3.3.



GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

2.2 GizmoDistribution fundamentals

The basic elements of GizmoDistribution are *clients, sessions, objects, events and attributes*. Objects, events and attributes are data elements. Clients are users of data and sessions are data spaces where data is published. The basic concept of GizmoDistribution is to distribute objects and events to clients on sessions.

2.2.1 Clients

A client is a user of distributed data. Clients have the capability to publish and subscribe for different types of events and objects on a session. A client can be both a publisher and a subscriber at the same time. Due to the data-centric design, clients do not need to know anything about where and if there are any other clients.

2.2.2 Sessions

A session is a data space representing a common client interest to share distributed data. Clients may join one or more sessions to publish and/or subscribe for different kinds of data. Sessions have a unique name and can be either local or global. A local session only exist within the local process, while a global session is shared between processes and hosts on a network. See also section 2.2.6.

2.2.3 Objects

An object is an instance of data stored on a session. It has a lifetime from the moment it is added until it is removed from a session. It has a unique name and a set of named attributes that are defined in run-time. The set of attributes is dynamic, i.e. it can be modified at any time. Subscribing clients are notified when objects are added, updated or removed from a session.

2.2.4 Events

An event is a temporary instance of data. It is not stored on a session; it only exists on the session the moment it is sent. Just like an object is has a set of named attributes that are defined in run-time, but the set of attributes is static, i.e. it can not be updated after the event has been sent. Subscribing clients are notified when an event is sent on a session.

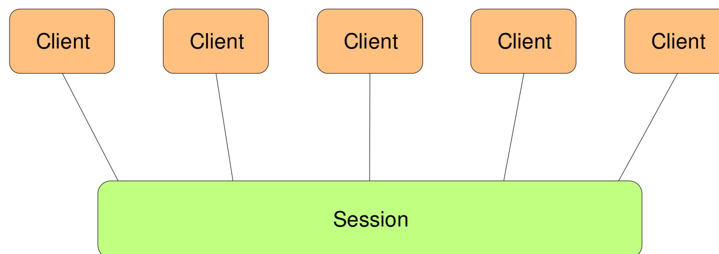
2.2.5 Attributes

Both events and objects have properties called attributes. An attribute basically consists of a name and a value. The name is unique per event or object. The value can be of many different types, but typically it is a string or a number.

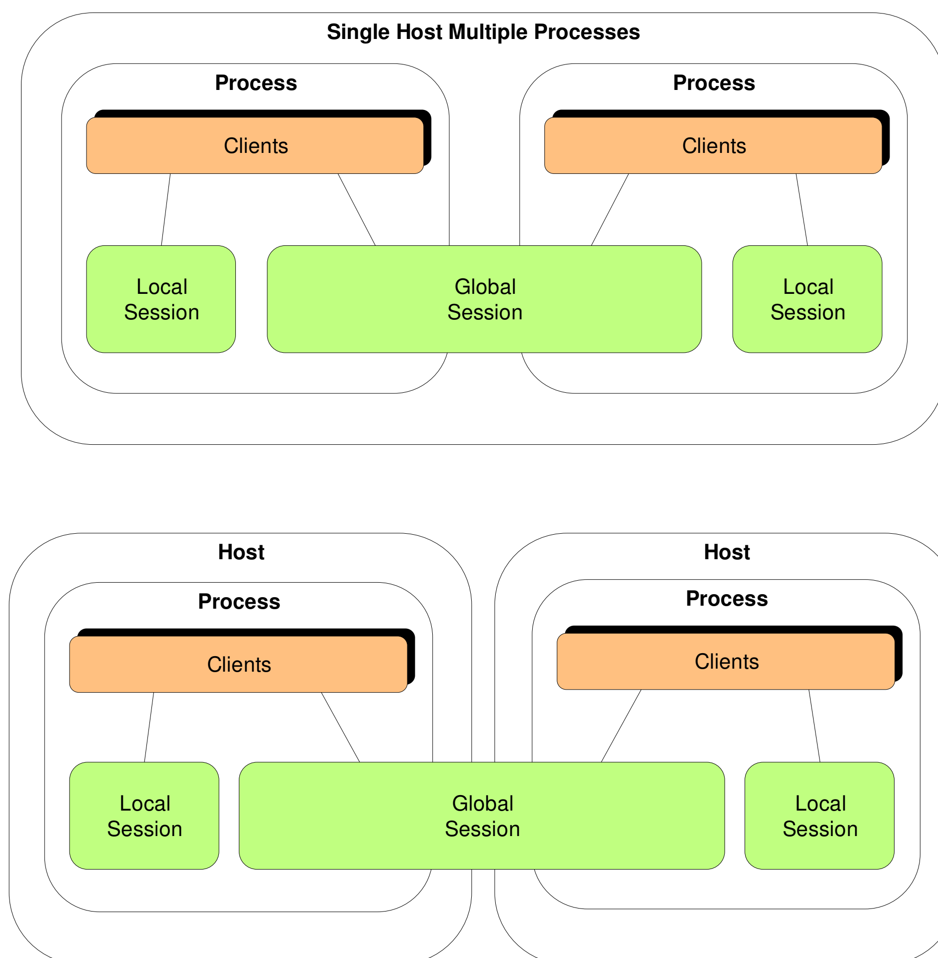
2.2.6 Distribution mechanism

GizmoDistribution allow clients to share objects and events on sessions in a uniform manner no matter where they are located.

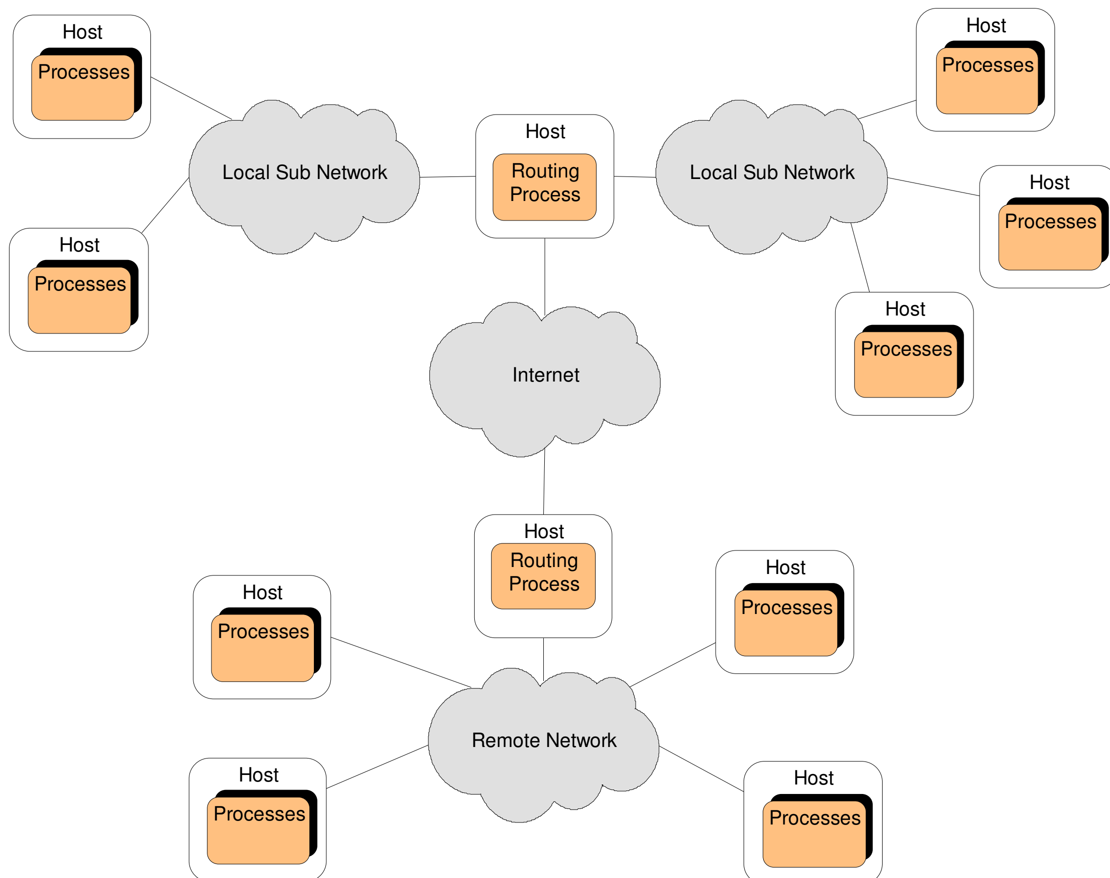
GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se



In the simplest case data is distributed only to clients within a single process (local sessions), but may also be distributed to different processes on the local host or to different hosts on a network (global sessions), as illustrated below. A global session may be shared by any number of processes. The remote distribution mechanism ensures that all clients joined to a session share consistent data in all of the processes. The remote distribution mechanism is further described in section 3.9.



A system can also be configured to distribute data between clients located on different networks, e.g. on the internet. See section 3.9.6 for details on how to route remote distribution between networks.



2.3 GizmoDistribution components

2.3.1 Libraries

GizmoDistribution include of two libraries:

- The **gzDistribution** library includes the basic framework of classes and utilities used for distribution of events and objects.
- The **gzRemoteDistribution** library includes classes used for enabling distribution of events and objects to other processes as described in section 2.2.6.

gzDistribution can be used separately from gzRemoteDistribution. There are dependencies only from gzRemoteDistribution to gzDistribution. The purpose of this is to let systems that do not use global sessions to exclude communication with remote process instances.

The libraries are dependent only of the gzBase library of the GizmoBase component in the GizmoSDK. No other external special-feature or third-party components are used.

2.3.2 Tools

Apart from the libraries GizmoDistribution also include three debugging and utility tools:

- **gzDistRouter** is a utility tool for routing/multiplexing distribution messages between different communication transports, e.g. connecting two sites over the internet.
- **gzDistMonitor** is a debugging tool that can be used for viewing distribution data (sessions, objects, events and attributes) and clients within a process.
- **gzDistSniffer** is a debugging tool used for viewing distribution messages sent on the network.

Graphical user interfaces are built with QT from Trolltech (http://www.trolltech.com/).

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

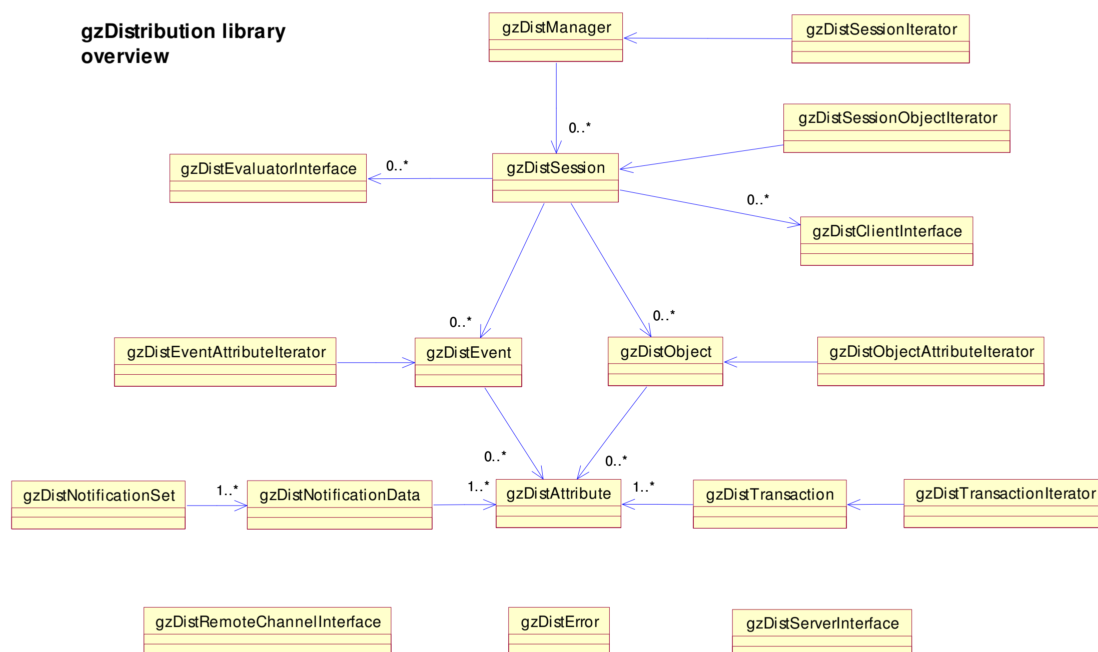
3 GIZMODISTRIBUTION DETAILS

This chapter describes the details of the SDK. The first section gives on overview and a brief introduction to the main classes of the API. The remaining sections describe different aspects of the API in more detail.

3.1 Classes

GizmoDistribution consist of two libraries; gzDistribution and gzRemoteDistribution, as described in section 2.3. This section gives a brief introduction to the classes that define the public interface of the libraries.

3.1.1 gzDistribution classes



The UML-diagram above shows the main classes of the gzDistribution library and their relationship.

`gzDistManager` is the kernel of GizmoDistribution that is responsible for local distribution. There is only one manager instance per process (singleton). The interface includes methods for general configuration as well as for starting and stopping the distribution framework. The manager also keeps track of all current clients and sessions in the local process. The manager is further described in section 3.2.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

gzDistSession represent sessions. Sessions has clients, store objects and distribute events. Sessions are either local or global. A local session only exist within the local process while global sessions are distributed to remote instances. Sessions are described in section 3.3 and 3.9.1.

gzDistClientInterface is the base class for clients to inherit. It is through this interface class that clients interact with the sessions and the data. Clients typically join one or more sessions, subscribe for objects and events and possibly add and update objects and send events. Clients are notified about subscribed changes on the sessions through callbacks. The client interface is described in section 3.6.

gzDistObject represent objects that can be added to sessions. An object instance can only belong to one session at a time. It is possible for clients to add, remove and update attributes on objects in run-time. Objects are described in section 3.4.

gzDistEvent represent a events that can be sent on sessions. An event instance can only be sent once and only on one session. Events are not stored on a session; they only exist on a session the moment they are sent. Events are described in section 3.5.

gzDistAttribute represent basic attributes. They have a name and a dynamic type value. Both objects and events are built up of attributes.

gzDistTransaction is used for grouping attributes together to perform a transaction on an object. A transaction is an atomic operation including one or more attributes to be updated added or removed from an object.

gzDistNotificationData is the attribute container for client object notifications.

gzDistNotificationSet is a collection of notification data. Clients receive instances of this class in notification callbacks.

gzDistEvaluatorInterface is the base class for evaluators to inherit. Evaluators are executed when objects are added, updated or removed from a session. The purpose is to accept or reject new objects, object removals and object transactions in order to keep the objects on a session consistent. A transaction will be canceled if it is rejected by any evaluator. An evaluator may update a transaction to make it valid. See also section 3.7.

gzDistServerInterface provide a mechanism to control where to execute certain functions (services) in a distributed environment. The server interface is described in section 3.8.

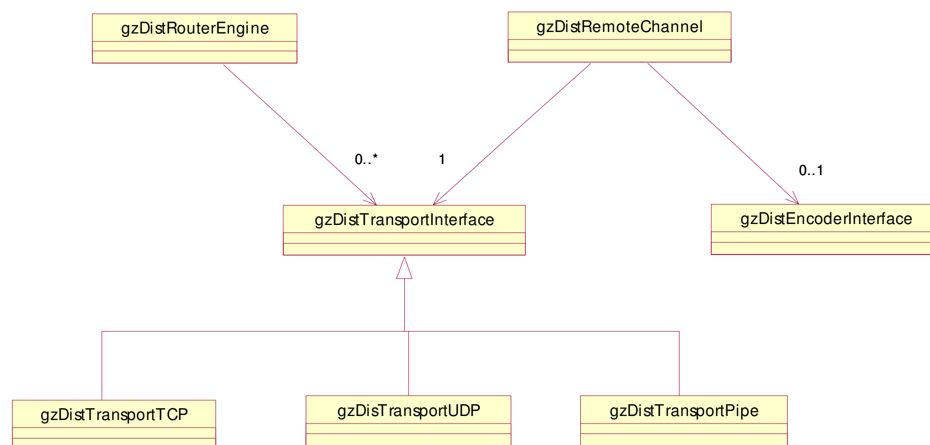
gzDistSessionIterator is used for iterating over sessions within the local process, **gzDistSessionObjectIterator** for iterating over objects in a session, **gzDistObjectAttributeIterator** for iterating over attributes in an object, **gzDistEventAttributeIterator** for iterating over attributes in an event, and **gzDistTrnasactionIterator** for iterating over attributes in a transaction.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

`gzDistError` contains error information. GizmoDistribution stores the last error on a per thread basis.

`gzDistRemoteChannelInterface` is an abstract class defining the interface for communication with remote instances. The interface is implemented by the `gzDistRemoteChannel` class in the gzRemoteDistribution library (see section 3.1.2).

3.1.2 gzRemoteDistribution classes



The UML-diagram above shows the main classes of the gzRemoteDistribution library and their relationship. The details of remote distribution are described in section 3.9.

`gzDistRemoteChannel` implements the interface for remote distribution. To enable remote distribution (i.e. global sessions) the `gzDistManager` instance must be initialized with two remote channels (the server channel and the session channel). Each channel need to have a transport for communication. See also section 3.9.2.

`gzDistTransportInterface` is a base class for distribution transports. Transports are used for transporting data between remote instances. They encapsulate different basic communication protocols (and media). Currently there are transports defined for UDP, TCP and named pipe communication. See also section 3.9.4.

`gzDistEncoderInterface` gives the possibility to enable custom encoding/formatting of messages sent on the network, e.g. encryption or compression. See also section 3.9.5.

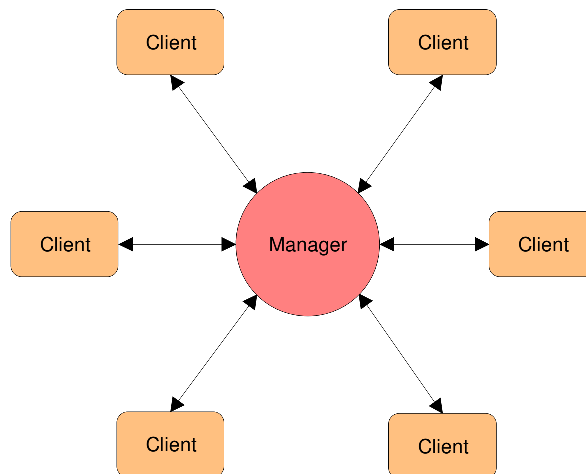
`gzDistRouterEngine` can be used for routing/multiplexing messages between different communication transports. See also section 3.9.6.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.2 Distribution manager

3.2.1 The kernel

The distribution manager `(class gzDistManager)` is the kernel of GizmoDistribution. There is only one singleton manager per process. It is the engine for distribution of data within the local process and keeps track of the current clients and sessions in the process. The internal manager thread runs the kernel of the distribution framework responsible for all local processing and notification. The manager serves as a hub and a filter for notifications to and from clients through sessions.



3.2.2 Starting and stopping

The manager interface includes methods for general configuration as well as for starting and stopping the distribution framework. The simplest configuration is for local distribution between clients within a single process.

```
// Create the manager
gzDistManagerPtr manager = gzDistManager::getManager(TRUE);

// Start local distribution
manager->start();

. . .

// Stop distribution
manager->shutDown();

// Release the manager
manager = NULL;
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

It is almost as simple to create a default configuration for global distribution. The difference is that two channels must be supplied; one channel for session communication and one for server communication. The code example below uses the default channel configuration.

```
// Create the manager
gzDistManagerPtr manager = gzDistManager::getManager(TRUE);

// Start global distribution using the default channels
manager->start(gzDistCreateDefaultSessionChannel(),
                gzDistCreateDefaultServerChannel());

. . .

// Stop distribution
manager->shutDown();

// Release the manager
manager = NULL;
```

If you do not want to use the default channels you will have to create and configure the channels your self. See section 3.9 for details about remote process communication and custom channel configuration.

3.2.3 Other tasks

Other tasks that are enabled through the manager interface are e.g.:

- custom client thread ticking (described in section 3.6.5)
- fast memory management configuration (described in section **Error! Reference source not found.**)
- local process debugging using gzDistMonitor (described in section 3.13)

Note also that the manager is reference managed (see section 3.10).

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.3 Sessions

3.3.1 Distributed data-space

A session `(class gzDistSession)` is a named data-space that represents a common client interest to share distributed data. A session has a number of joined clients subscribing and/or publishing data to the session. Data elements are either objects or events. Objects are named data stored on the session for clients to subscribe, add, update or remove. Events are not stored on the session, they only exists on the session the moment they are sent. For an event the session is more like an address where the event is sent.

Note that sessions are reference managed (see section 3.10).

3.3.2 Client operations

All main operations on a session should be done through the client interface (see section 3.6). Clients should always join a session before performing any operation. When the client no longer has any interest in the session it should resign from it. A session exists as long as there are clients joined to it. It is removed from a local process when the last local client resigns or when the manager is shut down, though it may still exist in other processes.

3.3.3 Sessions types

Sessions can be either *local* or *global*. On a local session data is only distributed locally to be shared only between clients in the local process. On a global session data is also distributed globally to be shared with clients in remote processes. A global session does not exist in a process until it has been created by a client inside that process. This means that data on a global session is the same in all processes where the session is created, but does not exist in processes where the session has not yet been created. If a session is created to be local in some processes but to be global in some, it will stay local where created local and global where created global, and they will not interfere. This is not recommended to do though, since it may confuse developers to make mistakes.

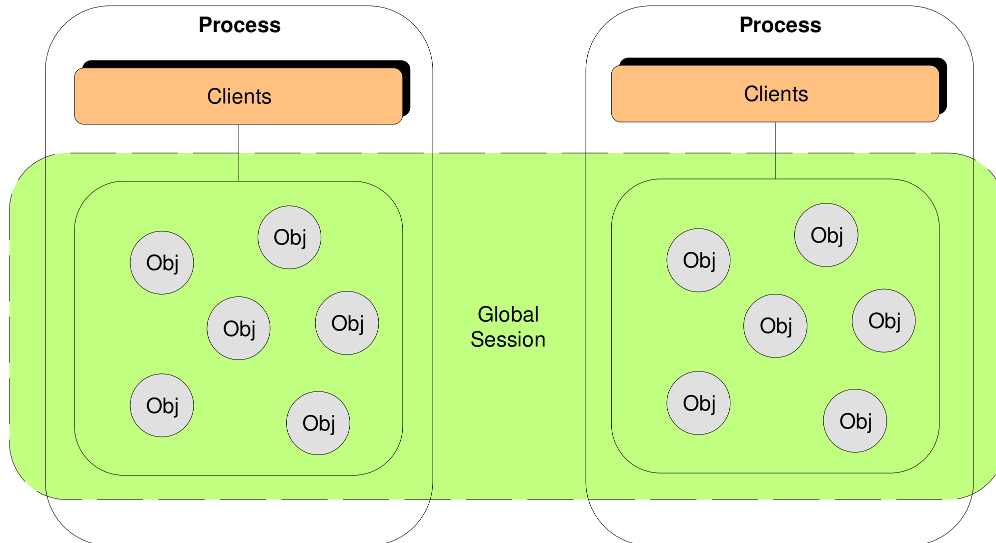
Local distribution

On a local session all operations and notifications to clients are performed by the local manager (see section 3.2). The manager is responsible for keeping objects consistent within the local process. Notifications are always delivered in the same order to all clients within the local process.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

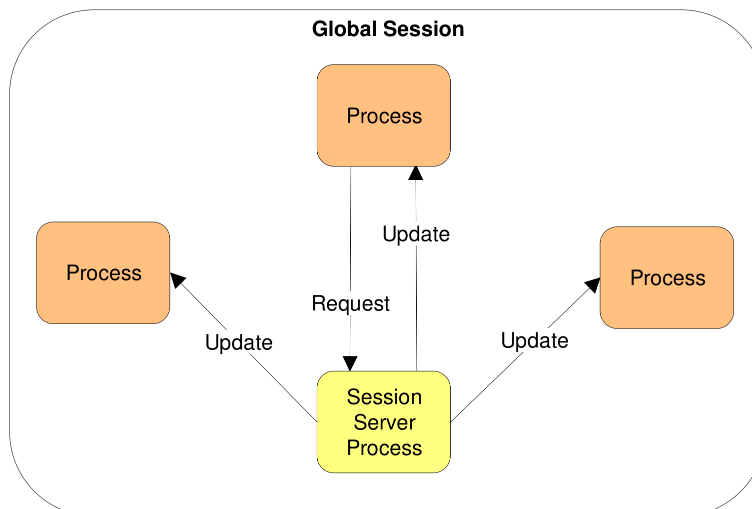
Global distribution

On a global session every process (where the session is created) keeps its own copy of all objects stored on it



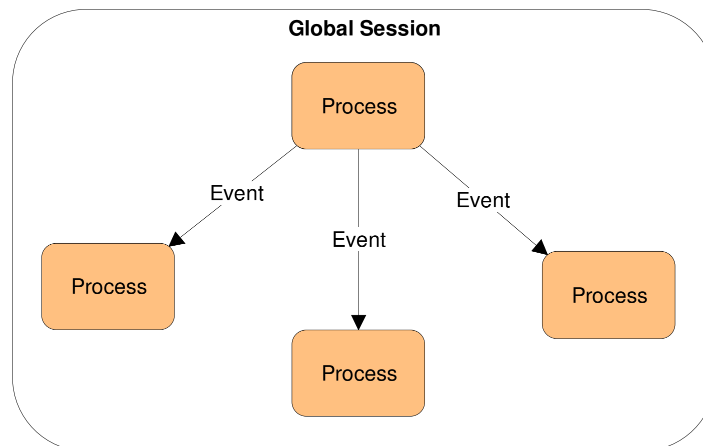
On a global session there is always one process acting as a server for that session (see also section 3.9.1). The session server is responsible for keeping the objects consistent with respect to attributes, attribute values and ownership, in all processes.

Operations on objects, like adding, updating and removing, are issued through the current session server, i.e. the operation is not performed in the local process, but instead a request is sent to the server. The server process requests from all clients and decide how and if the request should be executed and broadcast the result. The purpose is to make sure that objects on a global session are the same in all processes where the session exist, i.e. all clients in all processes has a consistent view of the objects. This also means that all object notifications are always delivered in the same order for all clients in all processes.



GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

Events on the other hand are not handled by the session server. This is not needed since they are not stored on a session and can not be changed after they have been sent. Instead events are broadcasted to other processes directly from the local process of the sending client. This means that event notifications are not necessarily delivered in the same order to clients in different processes. Though within a process events are always notified in the same order for all clients and events sent from one specific client are always notified to all other clients in the same order as they are sent.



See also section 3.9.1 for further details on global sessions and global distribution.

3.4 Objects

3.4.1 Data stored on sessions

An object `(class gzDistObject)` is data stored on a session with a unique name. An object can only belong to one session at a time, although different objects with the same name may be stored on different sessions. An object has a life time from the moment it is added to a session until it is being removed.

An object is built up of a set of named attributes. The set is dynamic, i.e. it can be modified at any time. Attributes may be added or removed and the values may be updated. The values can be of many different types (see `class gzDynamicType` defined in GizmoBase), but typically they are strings or numbers.

The attributes of an object may be owned by a client. It is only the owner that is allowed to update an attribute (see also section 3.6.4). Attributes that do not have an explicit owner are free for any client to update.

Note that objects are reference managed (see section 3.10).

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.4.2 Transactions

Attributes on an object can be added, updated or removed in two ways. Either attribute by attribute or a group of attributes in a transaction (`class gzDistTransaction`). A transaction is an atomic operation on an object including one or more attributes. Attributes belonging to a transaction will always be notified together to subscribing clients. If any of the attributes in the transaction is owned by another client the operation will be rejected.

3.4.3 Custom types

It is possible to create custom object types by inheriting the `gzDistObject` base class. This makes it possible for clients to subscribe for specific object types (see section 3.6.3).

```
// A custom object class definition (to be placed in your header-file)
class MyObject : public gzDistObject
{
      public:

            // Declare interface (used for subscription etc)
            GZ_DECLARE_TYPE_INTERFACE;

            // Construction and destruction
            MyObject(const gzString& name);
            virtual ~MyObject();

            // Clone interface
            virtual gzReference* clone() const;
};

. . .

// Type hierarchy information (to be placed in your cpp-file)
GZ_DECLARE_TYPE_CHILD(gzDistObject, MyObject, "MyObject");
```

If custom object types are used on global sessions there must be a factory installed for each type. Otherwise object will not be transferred between processes since the type is unknown to the distribution kernel. A factory is a name-less and empty instance of the object type that is registered at the manager before startup. This instance is cloned by the kernel when creating new objects.

```
// Install custom object type factory
manager->registerFactory(new MyObject(GZ_EMPTY_STRING));
```

Factories are automatically unregistered when the manager is shut down, but may also be manually unregistered if needed.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.5 Events

3.5.1 Temporary data

An event `(class gzDistEvent)` is a temporary instance of data sent on a session. An event has no name and no life time. It exists only the moment it is sent. An event is built up of a set of named attributes defined in run-time, just like objects, but the set is static, i.e. it can not be modified after the event has been sent. The attributes of an event has no owners since they can not be modified. Note that events are reference managed (see section 3.10).

3.5.2 Custom types

It is possible to create custom event types by inheriting the `gzDistEvent` base class. This makes it possible for clients to subscribe for specific event types (see section 0).

```
// A custom event class definition (to be placed in your header-file)
class MyEvent : public gzDistEvent
{
      public:

            // Declare interface (used for subscription etc)
            GZ_DECLARE_TYPE_INTERFACE;

            // Construction and destruction
            MyEvent();
            virtual ~MyEvent();

            // Clone interface
            virtual gzReference* clone() const
};

. . .

// Type hierarchy information (to be placed in your cpp-file)
GZ_DECLARE_TYPE_CHILD(gzDistEvent, MyEvent, "MyEvent");
```

If custom event types are used on global sessions there must be a factory installed for each type. Otherwise events will not be transferred between processes since the type is unknown to the distribution kernel. A factory is an empty instance of the event type that is registered at the manager before startup. This instance is cloned by the kernel when creating new events.

```
// Install custom event type factory
manager->registerFactory(new MyEvent);
```

Factories are automatically unregistered when the manager is shut down, but may also be manually unregistered if needed.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.6 Clients

3.6.1 Data user

Clients are the users of distributed data. Clients typically join one or more sessions and subscribe and add some subscriptions. They are then notified about subscribed changes as they appear through a virtual callback interface. Clients also have the capability to send events as well as to add, update and remove objects on the sessions they are joined to. In fact it is only from the client interface that this is possible.

3.6.2 Initailization

Clients are implemented by inheriting the `gzDistClientInterface` class and implementing the appropriate virtual notification callbacks. The client interface is the most important interface of the GizmoDistribution SDK. All main tasks are performed through this interface.

```
// A customized client class
class MyCustomClient : public gzDistClientInterface
{
	public:
		MyCustomClient(const gzString& name);
		virtual ~MyCustomClient();

	private:
		// Add appropriate callbacks as needed ...

};
```

Every client instance should be created with a name that is unique within the local process. Before joining any sessions the client must be initialized. A client is initialized with a thread pool id and a tick interval. Clients using the same pool id get notifications and ticks on the same thread (see section 3.6.5). When the client is uninitialized it will automatically be resigned from all sessions.

```
// Initialize client
gzDistClientInterface* client = new MyCustomClient("my_client");
client->initialize(myTickInterval, myThreadPool);

. . .

// Uninitialize client
client->uninitialize();
delete client;
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.6.3 Subscriptions and notifications

The table below gives an overview of available subscriptions and notifications and how they are triggered. All subscriptions can be canceled by calling a corresponding unsubscribe function. All subscriptions corresponding to a session are cancelled automatically when the client resigns from the session.

Subscription	Notification callback	Triggered by

`subscribeSessions()`	`onNewSession()`
	
	`onRemoveSession()`
		Session created (in the local process)
		
		Last local client resign

`subscribeEvents()`	`onEvent()`	`sendEvent()`

`subscribeObjects()`	`onNewObject()`
	
	`onRemoveObject()`
		`addObject()`
		
		`removeObject()`

`subscribeAttributes()`	`onNewAttributes()`
	
	`onRemoveAttribute()`
		`updateObject()`
		
		`removeAttributes()`
		`removeAttribute()`

`subscribeAttributeValue()`	`onUpdateAttributes()`	`updateObject()`

When subscribing for sessions a client will notified when sessions are created or removed within the local process, i.e. global sessions that are not created in the local process will not be notified.

Subscriptions for events and objects regard a specified type on a session. Clients will get notified when events of a subscribed type (or subtype) are sent and when objects of a subscribed type (or subtype) are added on the session.

There are two different kinds of attribute subscriptions. A client can subscribe for either new and removed attributes on an object (`subscribeAttributes()`) or for value updates of a specific attribute (`subscribeAttributeValue()`).

Object notifications are done by value, i.e. every update of an object is notified with the values resulting from the transaction it was triggered by. The purpose of this is to make sure that the clients get updated with correct states in consecutive order. Since notification callbacks are asynchronous this also means that the notification data (`class gzDistNotificationData`) may differ from the state of the actual object in a callback.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.6.4 Ownership

Clients may own attributes of objects. If an attribute is explicitly owned by a client no other clients are allowed to remove or update it.

Clients request ownership by calling the `requestOwnership()` method. When ownership is granted the client will be notified through the `onGrantOwnership()` callback. The current owner will be notified through the `onRequestOwnership()` callback when another client has requested ownership. To drop ownership or remove an ownership request the client should call `dropOwnership()`. There is also a way to force ownership to be moved from one client to another. This is done by calling `pullOwnership()`. This feature should be used with care.

3.6.5 Threading

All calls from the clients to the distribution kernel are asynchronous, i.e. they are executed on the manager thread instead of the calling client thread. Though in most cases is it possible to wait for the result by specifying a wait timeout.

Notifications to the clients are also asynchronous, i.e. they are not done on the manager thread, but on a thread specified when the client is initialized. A client is initialized with a thread pool id and a tick interval. Clients using the same pool id are notified on the same thread.

Optionally clients can get ticked (`onClientTick()` callback) on the very same thread at a desired interval. Clients may also decide not to use the thread pool, but instead trigger the notifications and ticks from a custom thread (e.g. a GUI-thread). There can be only one custom thread per process. Ticking is triggered from the manager using the method `processCustomThreadClients()`. The purpose of the tick callback is to make it possible to create single-threaded clients.

The purpose of the asynchronous calls and notifications is to enhance the efficiency and stability of the distribution kernel, and to prevent client deadlocks.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.7 Evaluators

The purpose of an evaluator is to accept or reject object transactions. If a transaction is rejected by any evaluator the transaction will be canceled. Evaluators are allowed to add attributes to a transaction to make it valid, but should not have any other side effects.

An evaluator class is created by inheriting the `gzDistEvaluatorInterface` class and implementing the appropriate evaluation callbacks. Note that evaluators are reference managed (see section 3.10).

```
class MyEvaluator : public gzDistEvaluatorInterface
{
      public:

            MyEvaluator(gzType* classType);
            virtual ~MyEvaluator();

      private:

            gzBool onNewObject(gzDistObject* object);
            gzBool onRemoveObject(gzDistObject* object);
            gzBool onUpdateObject(gzDistTransaction* transaction,
                                  gzDistObject* object);
            gzBool onRemoveAttributes(gzDistTransaction* transaction,
                                  gzDistObject* object);
};
```

An evaluator is registered to one or more sessions to evaluate a specific object type. The evaluator is then executed when an object is being added, updated or removed. Evaluators are only executed in the process where the transaction was initiated.

```
// Create an evaluator
gzDistEvaluatorInterfacePtr evaluator = new
    MyEvaluator(gzDistObject::getClassType());

// Register the evaluator to a session
session->registerEvaluator(evaluator);

. . .

// Unregister the evaluator
// Normally you don't need to bother since evaluators are
// unregistered automatically when sessions are destroyed
session->unregisterEvaluator(evaluator);
evaluator = NULL;
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.8 Servers

The purpose of the distributed server support in is to provide a mechanism to control where to execute certain server functions (services) in a network. A server function is uniquely identified by a function identifier (name). All server instances sharing a name are capable of executing the same function, but only one will be active at the time. When the active instance is shutdown, another instance will take over. The server distribution mechanism is priority based, i.e. the instance with the best capabilities (the highest priority) will be active while all the other instances are passive. If one or more instances have the same priority the one that was started first will be the active one.

The server is capable of recovering from a situation where the active server crashes or is disconnected from the network. It will be discovered by the other instances when ping messages no longer are received, and after a timeout period of 10 seconds, the remaining instance with the highest priority will take over as active.

The server is also capable of recovering from situations where there for some reason is more than one active instance, e.g. when networks have been disconnected and are reconnected again. When an active instance discovers that there is another active it will go back to passive if it has lower priority.

The distributed server support is used by inheriting the `gzDistServerInterface` class and implementing the virtual callbacks.

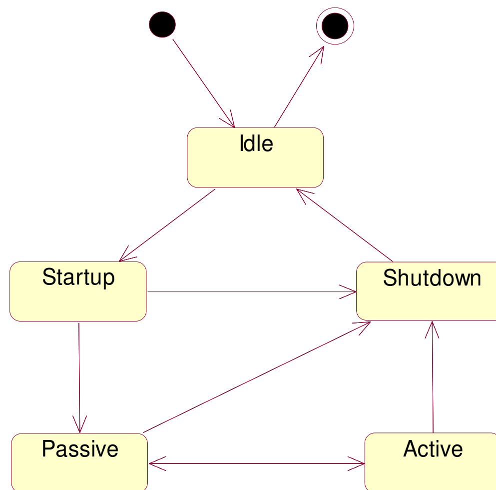
```
// A customized server class
class MyCustomServer : public gzDistServerInterface
{
	public:
		MyCustomServer();
		virtual ~MyCustomServer();

	private:

		// Add appropriate callbacks as needed

};
```

The global session server (see section 3.3.3) uses the same basic implementation as the public server interface.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.8.1 States

A sever instance can have five different states as shown in the state chart below.



The two main states are **passive** and **active**. Only the instance currently running the server function is allowed to be in the active state. All other instances are passive. The **startup** and **shutdown** states are temporary states entered at startup and shutdown of an instance. Disabled instances are in the **idle** state.

3.8.2 Priorities

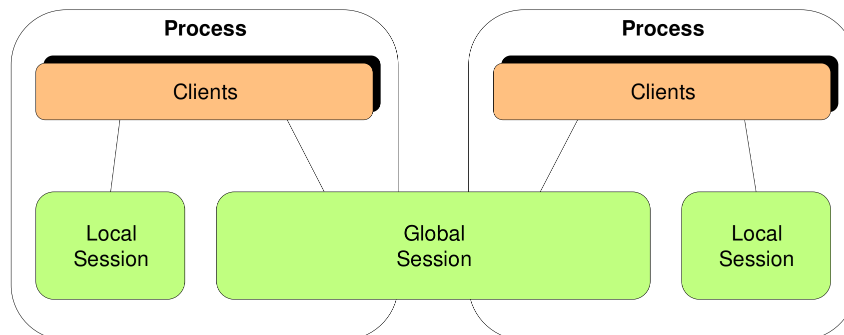
Five different basic server priorities are defined with **max** as the highest and **very low** as the lowest. There is also the special priority called **never**, that can be used by instances that don't want to run the server function, but yet wants to inform other instances of its presence.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

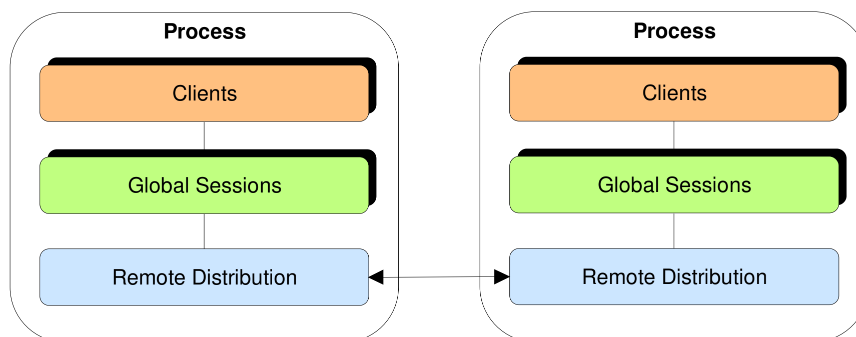
3.9 Remote Distribution

3.9.1 Sessions and servers (revisited)

The fundamental concept of GizmoDistribution is to let clients share objects and events on sessions. On a local session only clients with in the local process share data, while on a global session clients share data no matter where they are located.

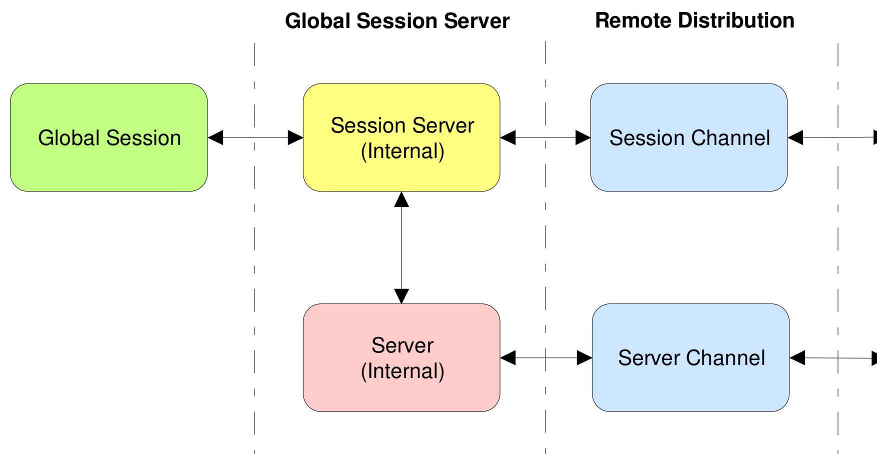


Global sessions use a remote distribution mechanism to share objects and events. Every process where a global session is created keeps its own copy of all objects stored on it. The remote distribution mechanism is responsible for transferring the data between processes. The same mechanism is used both between processes on the local host and between different hosts on a network.



Two different communication channels are used for remote distribution; the *session channel* and the *server channel*. The session channel transfer session data (objects and events, etc) and the server channel transfer server control data (see also section 3.8). The reason for the two channels is that servers require control data to be delivered in time, i.e. delayed messages may cause errors. Therefore different channels are used to avoid situations where heavy session data transfer delay server data. The server channel is internally handled with a higher priority than the session channel.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se



On a global session there is always a session server responsible for keeping the objects consistent in all processes. The session server exists in all processes where the session is created but is only active in one of the processes. If the current active session server is stopped or crashed, another process will take over. This is implemented using the server mechanism described in section 3.8. Hence global sessions use both the session channel and the server channel.

3.9.2 Channels

GizmoDistribution use two communication channels (`class gzDistRemoteChannel`) as described in section 3.9.1. The channels are responsible for sending and receiving as well as for encoding and decoding messages. Each channel is associated with a transport that provides the underlying communication mechanism (see section 3.9.4) and may optionally also be associated with a custom encoding mechanism to add e.g. compression and/or encryption of data sent over the network (see section 3.9.5). It is important that the two channels do not have the same transport configuration; otherwise they will interfere with each other.

```
// Setup session channel
gzDistRemoteChannel* sessionChannel = new gzDistRemoteChannel;
sessionChannel->setTransport(sessionTransport, sessionEncoder);

// Setup server channel
gzDistRemoteChannel* serverChannel = new gzDistRemoteChannel;
serverChannel->setTransport(serverTransport, serverEncoder);

// Start global distribution
manager->start(sessionChannel, serverChannel);
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

The above code example shows how to create channels using custom transports. For convenience there are two functions provided for creating channels with default settings. `gzDistCreateDefaultSessionChannel()` can be used for creating a default session channel and `gzDistCreateDefaultServerChannel()` for creating a default server channel. `gzDistCreateChannel()` is a generic helper function for creating channels without having to bother about the transport classes.

3.9.3 Transfer modes

Communication may be configured for either *reliable* or *best-effort* data transfer mode. Best-effort mode incurs the least overhead, but does not provide reliable communication, i.e. messages may be lost on the network occasionally. Best-effort is the default mode and should be good enough for most applications. For networks including unreliable links or links with limited bandwidth it is recommended to use the reliable mode.

To enable reliable transfer mode you should supply the constructor of your channel with a buffer size. The buffer size states the maximum number of messages to be stored in a re-send buffer. A zero buffer size will give best-effort mode

```
// Create reliable mode channel
gzDistRemoteChannel* channel = new gzDistRemoteChannel(bufferSize);
```

The buffer is a FIFO where outgoing messages are stored. When a remote process discovers that one or more messages have been lost, it will request for those messages to be re-sent. Received messages are acknowledge periodically. It is recommended to use the default buffer sizes. Note that too small buffers may decrease the throughput performance.

Note that all processes in a system using GizmoDistribution must use the same transfer mode configuration. A process configured for reliable mode can not cooperate with a process configured for best-effort mode and vice versa.

3.9.4 Transports

Transports handle communication with remote processes using different transport protocols. Currently there are transports defined for *UDP*, *TCP* and *named pipe* communication. The default channels use UDP multicast or UDP broadcast. These are the common transport types used on local area networks. Other transport types are primarily aimed for situations where you need to connect different remote network or to limit the communication to processes on the local host. How to connect and route/multiplex communication between different transports is described in section 3.9.6.

All transports are datagram-oriented (like UDP), even though the underlying protocol may be stream-oriented (e.g. TCP and named pipe). The reason is that it must be possible to recognize complete messages (or message fragments) from different sources even though they might be interleaved.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

UDP

An UDP transport (`class gzDistTransportUDP`) can be configured for unicast, multicast or broadcast. Multicast and broadcast are the commonly used transports since they make it possible to distribute data between any number of hosts on a local area network without wasting network bandwidth. Unicast is useful only for connecting two single processes or for connection networks or network segments not reachable by multicast or broadcast.

TCP

The TCP transport (`class gzDistTransportTCP`) can be configured to be either a server listening on a given port or a client connection to a given port on a given host. The server may accept only one client connection at a time. Like UDP unicast the TCP transport is useful for connecting networks or network segments not reachable by multicast or broadcast. The purpose of the TCP transport is to provide a robust connection when there are unreliable links or links with limited bandwidth. TCP is more efficient than the reliable transfer on those links.

Named pipe

A named pipe transport (`class gzDistTransportPipe`) can be either local or global and a server or a client. A local named pipe stays within the local host, while a global named pipe is accessible on every host on a local network. The server transport creates the pipe, while the client connects to an existing pipe. The main purpose of the named pipe transport is to make it possible to distribute data between processes on the local host only with a local pipe. This can be especially useful on Win32 hosts that don't have any physical network or loopback adapter installed.

3.9.5 Custom encoding

The custom encoding interface (`class gzDistEncoderInterface`) gives the possibility to enable custom encoding/formatting, e.g. encryption or compression, of network data. A custom encoder is created by inheriting the `gzDistEncoderInterface` class and implementing the virtual protected `onEncode()` and `onDecode()` methods. `onEncode()` is called when an outgoing message shall be encoded and `onDecode()` is called when an incoming message shall be decoded.

```
// A custom encoder class
class MyCustomEncoder : public gzDistEncoderInterface
{
	public:
		MyCustomEncoder(gzDistEncoderInterface* parent = NULL);
		virtual ~MyCustomEncoder();

	protected:
		gzUInt onEncode(const gzUByte* source, gzInt sourceLength,
		                          gzArray<gzUByte>& destination);

		gzUInt onDecode(const gzUByte* source, gzInt sourceLength,
		                          gzArray<gzUByte>& destination);
};
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

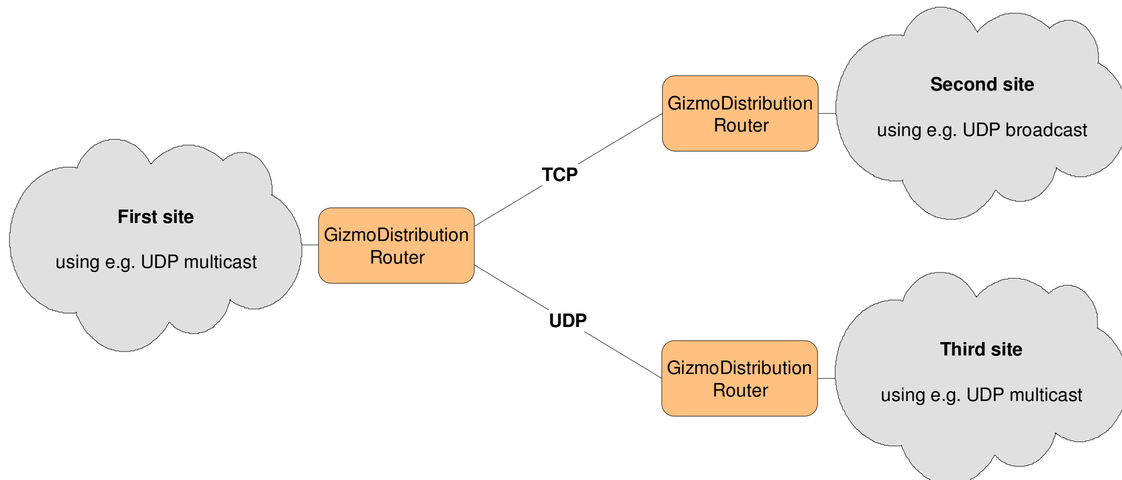
It is possible to chain encoder together by supplying a parent encoder to the constructor. Parent encoders are executed before the child when encoding and after the child when decoding.

GizmoDistribution supply two default encoders; one compressor and one simple scrambler. The compressor (`class gzDistDefaultCompressor`) uses an enhanced Ziv-Lempel algorithm. The scrambler (`class gzDistDefaultEncoder`) is a simple key-based encoder that provides basic message privacy. Note that it is not a qualified cipher and should not be used when security is an important issue.

Note also that using any type of encoders or scramblers will affect the performance. The max rate for sending and receiving data from the network will decrease.

3.9.6 Routing / multiplexing

The purpose of routing/multiplexing of messages between communication transports is to connect two remote network sites over e.g. the internet. Typically the two sites will run GizmoDistribution using UDP multicast or broadcast, but for being part of the same system they must be connected e.g. using a TCP transport or a UDP unicast transport. In the figure below show an example where three network sites are connected.



Router engine

The `gzDistRouterEngine` class provides the means for routing messages between different transports. The router engine will forward an incoming message from one transport to all other installed transports. The engine can be configured to route all incoming messages or only GizmoDistribution messages. Note that for GizmoDistribution to work properly there must be two different router engines; one for the session channel and one for the server channel (see section 3.9.2).

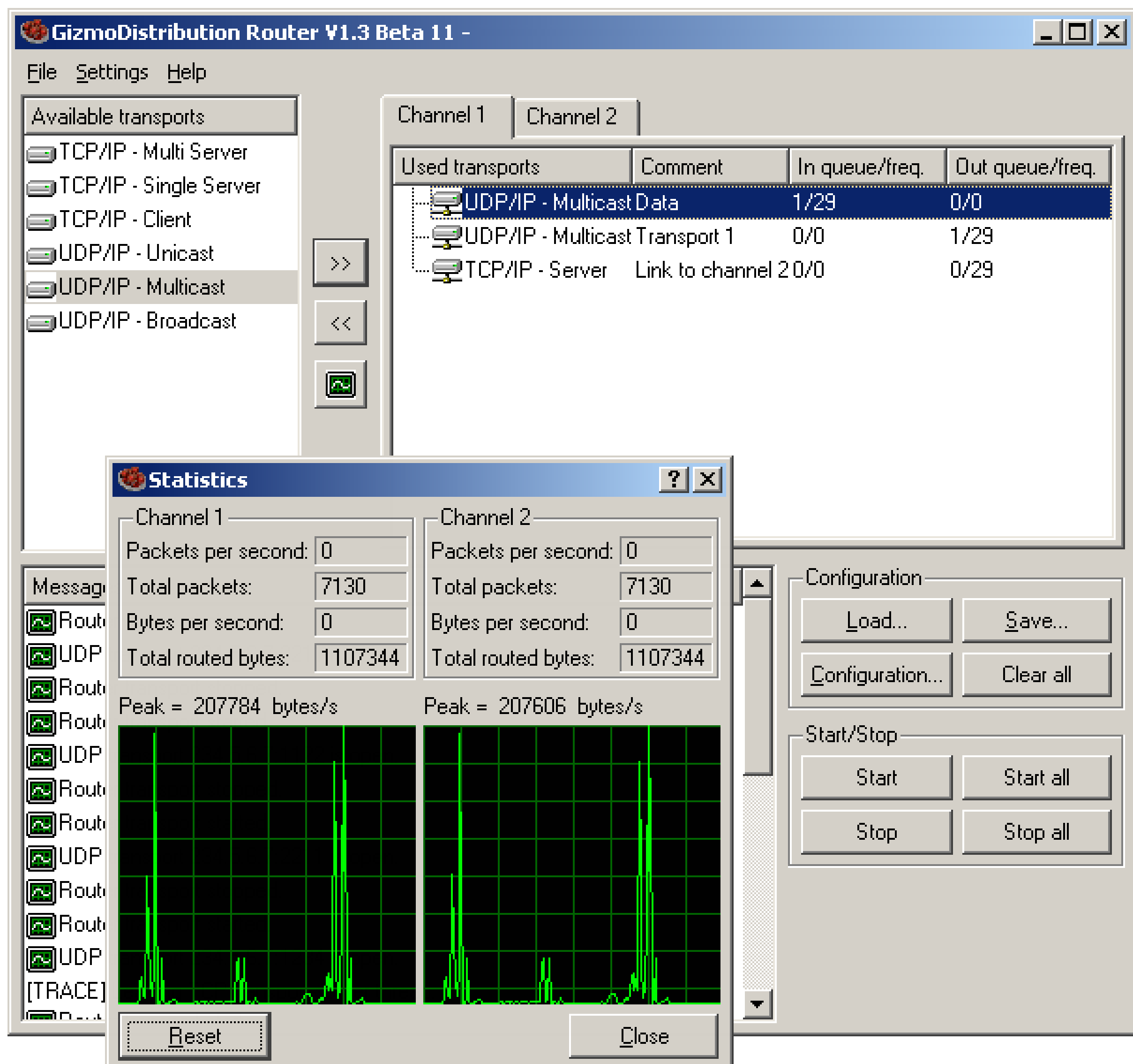
GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

```
// Create the router (started automatically)
gzDistRouterEngine* router = new gzDistRouterEngine;

// Add transports
router->addTransport(firstTransport);
router->addTransport(secondTransport);
router->addTransport(thirdTransport);

. . .

// Delete the router (stopped automatically)
delete router;
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

Router tool

The SDK also include a pre-compiled tool with a GUI (gzDistRouter) that implements a transport router/multiplexer. The tool implements two routers, one for each channel. It also includes communication configuration and statistics views.



GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.10 Reference management

GizmoDistribution uses reference management for data that is shared between local clients. All classes that inherit the `gzDistThreadSafeReference` class (e.g. `gzDistEvent` and `gzDistObject`) implements a thread-safe reference management interface with a `ref()` and an `unref()` method. The `ref()` method increments a reference counter starting from zero and the `unref()` method decrements the counter. When the counter reaches zero on a call to `unref()` the data is automatically deleted.

GizmoDistribution include pre-defined smart-pointer types for the reference managed classes. These are named `*Ptr`, e.g. `gzDistEventPtr` is a smart-pointer for the `gzDistEvent` class. A smart-pointer variable allocated on the stack will automatically call `ref()` and `unref()` within its scope when initialized with a reference managed instance.

See also the GizmoBase Programmers Manual for details about reference management.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.11 Iterators

There are iterators defined for iterating over;

- sessions in the local process (`gzDistSessionIterator`)
- objects in a session (`gzDistSessionObjectIterator`)
- attributes of an object (`gzDistObjectAttributeIterator`)
- attributes of an event (`gzDistEventAttributeIterator`)
- attributes of a transaction (`gzDistTransactionIterator`)

It is important to be aware that the iterators may block the manager thread. Therefore it is important use them fast and in as small scopes as possible. When frequent iterations are required it is more efficient to subscribe for the needed data and keep a local list or dictionary to iterate over.

3.12 Error management

GizmoDistribution manages errors on a per thread basis. The last error for every thread (both internal and external) interacting with any of the libraries is stored.

Information about the last error on the current thread can always be looked up with the `gzDistGetLastError()` function. By calling `gzDistReportLastError()` a description string of the last error may also be sent using the message mechanism of GizmoBase. The last error can be cleared by calling `gzDistClearLastError()`.

Error information is encapsulated by the `gzDistError` class. The error information includes e.g. an error level, an error code, an error description string and optionally an additional description string.

3.13 Debugging

3.13.1 gzDistMonitor

gzDistMonitor is a debugging tool that can be used for viewing data (objects, events and attributes) and clients on sessions within a process. Debugging of a process must be enabled from within the process. This is done by calling the `enableDebug()` method of the `gzDistManager` class as shown below.

```
// Enable debugging
gzDistManager::getManager()->enableDebug(TRUE);

. . .

// Disable debugging
gzDistManager::getManager()->enableDebug(FALSE);
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

3.13.2 gzDistSniffer

gzDistSniffer is a debugging tool used for viewing distribution messages sent on the network. The primary purpose is to enhance debugging of the GizmoDistribution remote distribution protocol in it self, but the gzDistSniffer can also be used for monitoring the network activity of processes or debugging the network architecture of a system using GizmoDistribution.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

4 TUTORIALS

This chapter contains step-by-step tutorials.

4.1 How to create GizmoDistribution applications

4.1.1 The GizmoChat application

This tutorial will explain how to create a simple GizmoDistribution application step-by-step. We will start with the example application called "GizmoChat", that can be found among the examples in GizmoDistribution online documentation (see http://www.gizmosdk.com/) and in the examples directory of the SDK installation. The example illustrates a simple application that sends and receives messages between one or more hosts on local network.

After we have setup the project environment as described in 4.2, we can start writing the code.

First of all we need to include all header files needed for GizmoDistribution. That is done through the `gzDistributionLibrary.h` file. So our first line will be:

```
#include "gzDistributionLibrary.h"
```

Now we want to be some kind of user of GizmoDistribution. We want to publish data and we want to receive data from some kind of global scope. To fulfil that, we have to be a client to the distribution system. That is simply done by inheriting `gzDistClientInterface()`. In the example, we have chosen to name our class `CChatClient`.

```
class CChatClient : public gzDistClientInterface
{
    public:
        // Constructor declaration.
        CChatClient();

        // Destructor declaration.
        virtual ~CChatClient();

        // The run method.
        gzVoid run();

    protected:

        // Virtual callback reimplemented from gzDistClientInterface.
        gzVoid onEvent(gzDistEvent* event);
};
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

The client class consists of a constructor, a destructor, a method named `run()` and the `onEvent()` method that is re-implemented from the base class. All we can see here is that it brings a pointer to a `gzDistEvent`. We'll talk more about this method later.

Let us move on to the implementation of the chat program. First we need a body for the constructor. It looks like this:

```
CChatClient::CChatClient() : gzDistClientInterface("ChatClient")
{
}
```

The only thing that is worth to mention here is that we need to give the client (our interface) a name. We have chosen the name "ChatClient".

The body of our destructor is empty.

```
CChatClient::~CChatClient()
{
}
```

Now let us have a look at the `run()` method.

```
gzVoid CChatClient::run()
{
    // Create the distribution manager.
    gzDistManagerPtr manager = gzDistManager::getManager(TRUE);

    // Start the manager and use default channels.
    manager->start(gzDistCreateDefaultSessionChannel(),
                   gzDistCreateDefaultServerChannel());
```

First we declare a reference handle to `gzDistManager`. We get the manager with the call `gzDistManager::getManager(TRUE)`. The argument means that the manager should be created if it not exists.

The next thing is to start the manager. The `start()` method on the manager takes two arguments. The arguments are two channels for communication with remote processes. We can skip the arguments if we just want to communicate inside our own process. In this case when we want to communicate outside our process we need channels to talk to other processes through. One channel is for transporting all user data like events and objects, and one channel is for system control communication. This is normally nothing you need to care about; it is just the way it works.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

There are two ways to create the channels. Either we do it manually by specifying what type of transport, which IP-address and port number we want to use, or we can shoes to use the default channels. In this case we do the simplest thing, we shoes the default channels. All this should be done only once per process. See the online documentation for details about the default channels.

To be able to work with the distribution system we need to be initialized. This is simply done with the following line:

```
// Initialize my distribution interface.
initialize();
```

We have the possibility to setup a tick interval if we want to do something on a regular interval (every x-th second). We do not need it here so we leave it blank.

Now we need something we to send and receive our messages on; something that delivers messages between clients. In GizmoDistribution this is called a session. A session has a name and can be either local or global. The name is a unique identification of the session. A local session distributes data only inside a process, even if we have installed channels. A global session distributes data outside the process but require installed channels. Let us go on and create the session. In this example we have chosen to give the session the name "ChatSession".

```
// Create a global session where all chat messages will be sent.
gzDistSessionPtr myChatSession = getSession("ChatSession", TRUE, TRUE);

// Join the session.
joinSession(myChatSession);
```

Here we declare a reference handle to the session, and get the session with the `getSession()` method. The first argument is the name of the session we want to have, the second tells if we want the session to be created if it does not exist (TRUE = create), the third argument tells that the session should be a global session (TRUE = global).

After we have a handle to the session we want to tell that we want to use the session. That is done with the `joinSession()` method. If the client does not join the session, it can not receive anything from it.

After that we need to tell the session that we want to receive data from it. In this example we are interested only in `gzDistEvent`. It is possible to create and use customized event types, but in this case we are satisfied with the base type. So we simply subscribe for all events of type `gzDistEvent` the session "ChatSession".

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

```
// Subscribe for all events on myChatSession.
subscribeEvents(gzDistEvent::getClassType(), myChatSession);
```

The `getClassType()` return run-time class type information (see also the GizmoBase programmers manual) used for identifying subscriptions.

Next we want to manage user input. What it does is to allocate buffers for alias and message and ask the user to enter an alias and messages to send to other users.

```
// Alloc a buffer for alias.
gzChar alias[128];

// Tell the user to enter his/her alias.
printf("Enter your alias:");

// Get user alias.
gets(alias);

// Print some information and a '>' at cursor position.
printf("Write your messages here. Type 'exit' to exit.\r\n>");

// Alloc a message buffer.
gzChar message[1024];

// Repeat until 'exit' is entered.
while (gets(message))
{
    // Quit if the user type 'exit'.
    if (gzString(message) == "exit")
    {
        // Break the 'while loop'.
        break;
    }

    // Don't send an empty message.
    if (strlen(message) == 0)
    {
        // Empty message.
        //Put a cursor at cursor position and wait for new input.
        printf(">");
        continue;
    }

    // Put a cursor at cursor position.
    printf(">");
```

The next step is to create a message, i.e. a `gzDistEvent` and put the message in it and send the message on our session.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

```
// Create a new event.
gzDistEvent* event = new gzDistEvent;

// Put the alias in the event.
event->setAttributeValue("Sender", alias);

// Put the message in the event.
event->setAttributeValue("Message", message);

// Put an id for current process in the event.
event->setAttributeValue("Id", gzDistGetCurrentProcessID());

// Send my message on myChatSession.
sendEvent(event, myChatSession);
```

The first line creates the event. The second and the third put the alias and the message in the event. The fourth line puts an ID of current process in the event. We do this because we want to know if the event was sent from our selves in the `onEvent()` method. An event consists of a number of attributes. An attribute consists of a name and a value. The name is a string and the value can be any type, typically a number or a string. To set an attribute to an event we use the `setAttributeValue()` method. After we have put all information in the event we want to send we send the event on our session. This will be repeated until the user write “exit” on the command line.

When we are tired on sending messages to everyone we release our session handle by setting it to NULL, we un-initialize our client interface and shut down the manager.

```
// Release my reference to the chat session.
myChatSession = NULL;

// Uninitialize my distribution interface.
uninitialize();

// Shut down the manager.
manager->shutDown();
```

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

Now, let us move on to the `onEvent()` method. This is a callback method that is inherited from `gzDistClientInterface`. We will be notified on all events that we have subscribed for in this function. The argument is a pointer to the event that is being sent to us.

```
gzVoid CChatClient::onEvent(gzDistEvent* event)
{
    gzDouble id;

    // Get the id from the event
    if (event->getAttributeNumber("Id", id))
    {
        // Don't receive my own messages.
        if (id == gzDistGetCurrentProcessID())
        {
            // I'm the sender -> Don't display.
            return;
        }
    }

    // Get the sender name
    gzString sender = event->getAttributeString("Sender");

    // Get the message
    gzString message = event->getAttributeString("Message");

    // Display the message.
    // To make this example to work on other platforms,
    //just change this line to a command that fit your platform.
    MessageBox(NULL, message, sender, MB_OK);
}
```

The first thing we do in this method is to check if the message was sent from my self. If this is the case, we just return and do nothing. If the sender wasn't me, we get the sender (the alias) and the message from the event and we display a message box with the alias and the message.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

The last thing to describe is the main routine. It simply creates an instance of our chat client and run it.

```
int main(int argc, char* argv[])
{
    // Create the chat client.
    CChatClient chatClient;

    // Run it!
    chatClient.run();

    return 0;
}
```

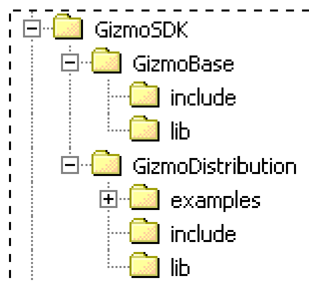
GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

4.2 How to setup GizmoDistribution projects

4.2.1 Microsoft Visual Studio 6

This tutorial explains step-by-step how to setup a GizmoDistribution project in Microsoft Visual Studio 6.

1. Unzip the GizmoDistribution "Libraries and Headers" file for "Win32 VS 6" and GizmoBase "Libraries and Headers" file for "Win32 VS 6" to "C:\". These files can be downloaded from http://www.gizmosdk.com/. If you extract the files using WinZip, it will automatically create a folder called "GizmoSDK".



2. Start **Microsoft Visual Studio 6**.

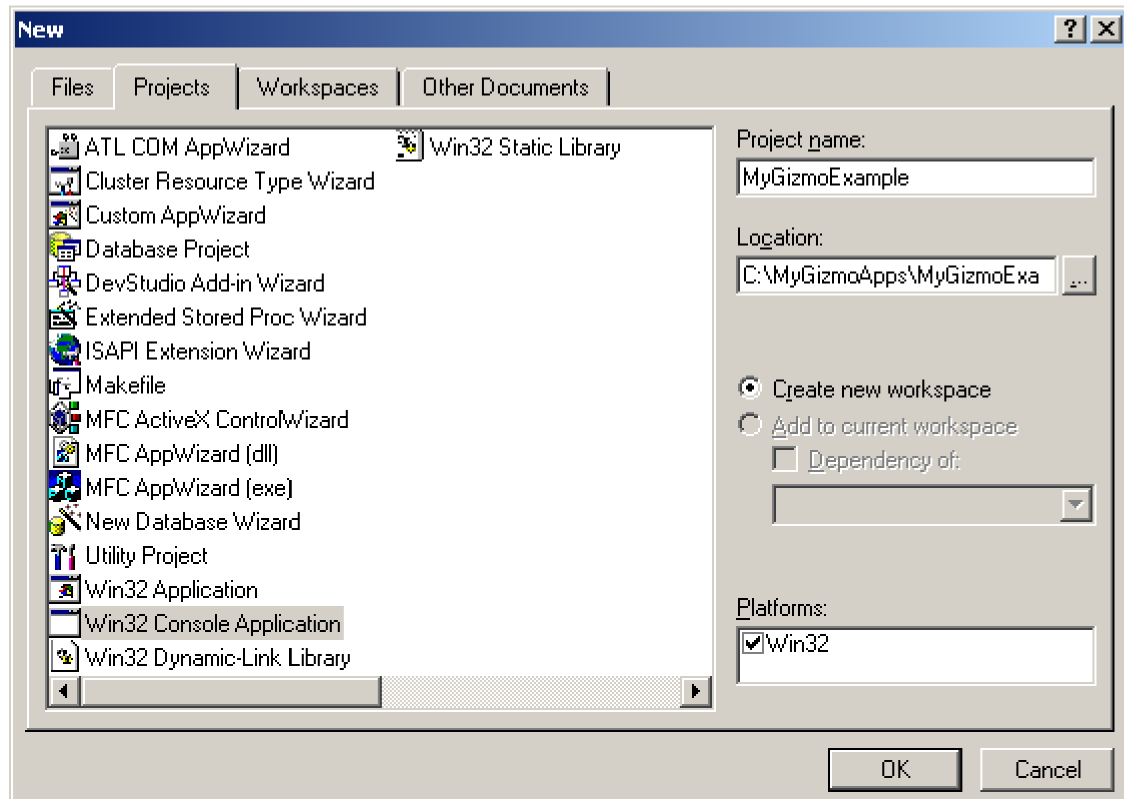
3. Go to the **File** menu and select **New…**

4. Select the type of application you want to create in the "Projects" tab. In this tutorial we select the **"Win32 Console Application"** alternative.

5. Select a project location in the "Locations" edit box. In this tutorial we chose **"C:\MyGizmoApps\"**.

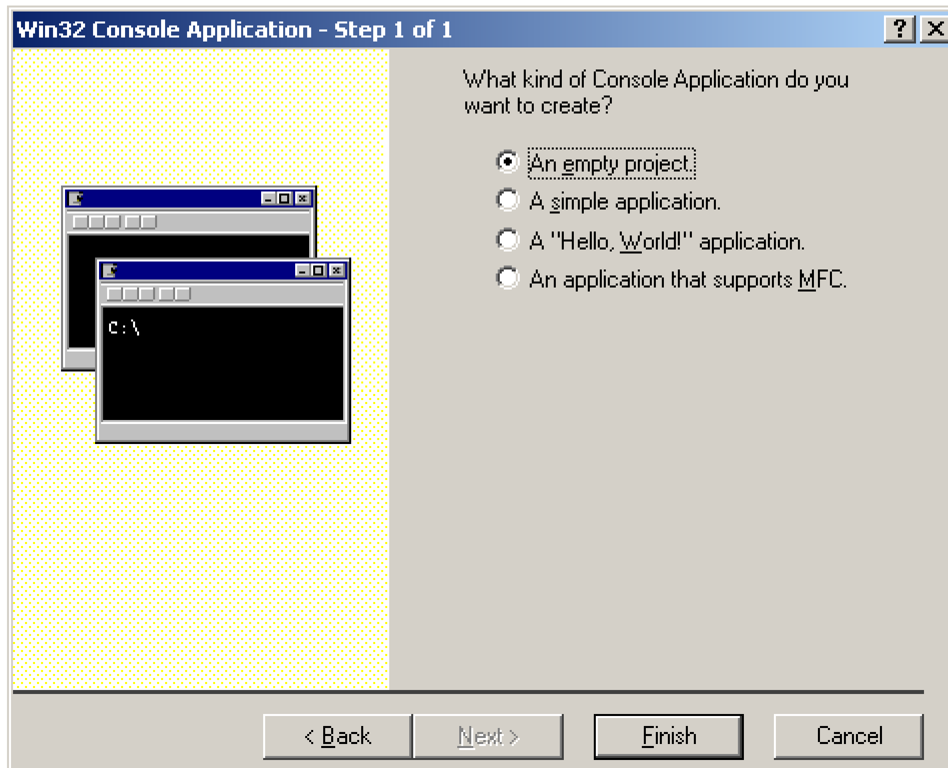
6. Select a project name in the **"Project name"** edit box. In this tutorial we have chosen to call our application "**MyGizmoExample**".

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se



7. Press **<OK>**

8. In the "Win32 Console Application" dialog, ensure that **"An empty project"** is selected and press **"Finish"**.



9. Press **<OK>** in the “New Project Information” dialog.

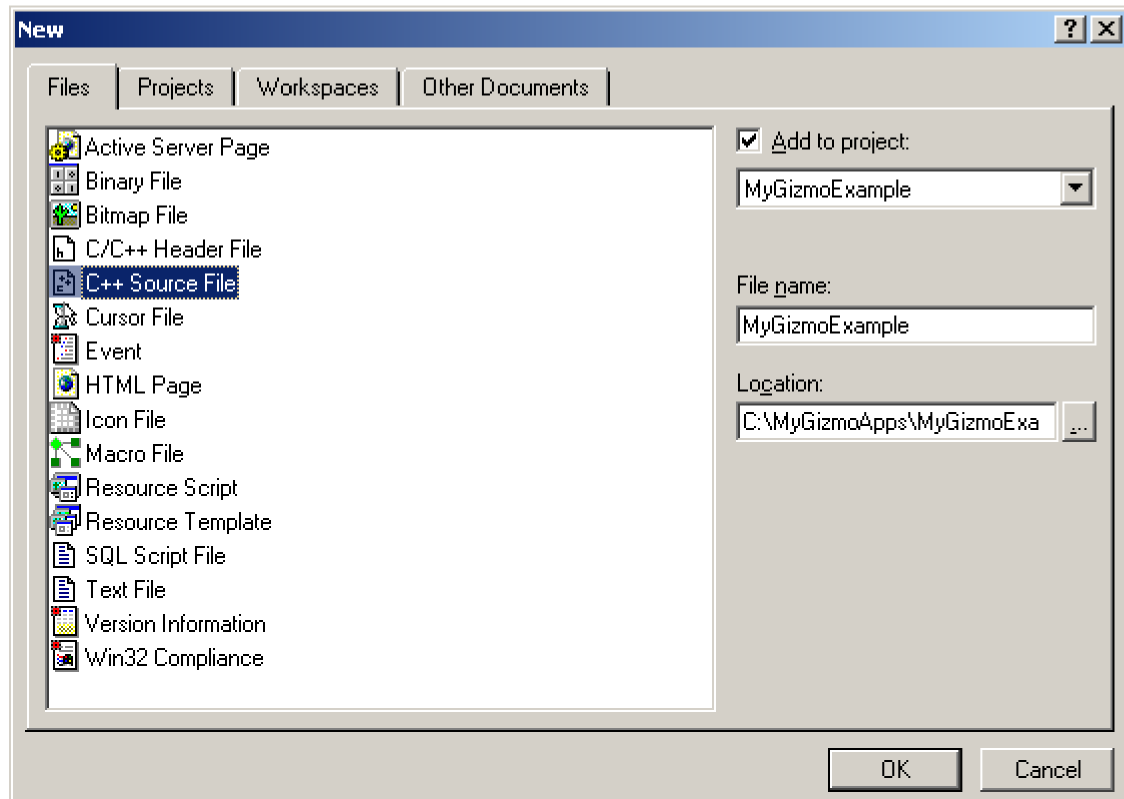
10. Go to the **File** menu and select **New…**

11. Ensure the **“Files”** tab is selected.

12. Select **“C++ Source File”**.

13. Ensure the **“Add to project”** checkbox is selected.

14. Enter **“MyGizmoExample”** in the “File name” edit box.



15. Press **<OK>**.

16. Go to the **Project** menu and select **Settings…**

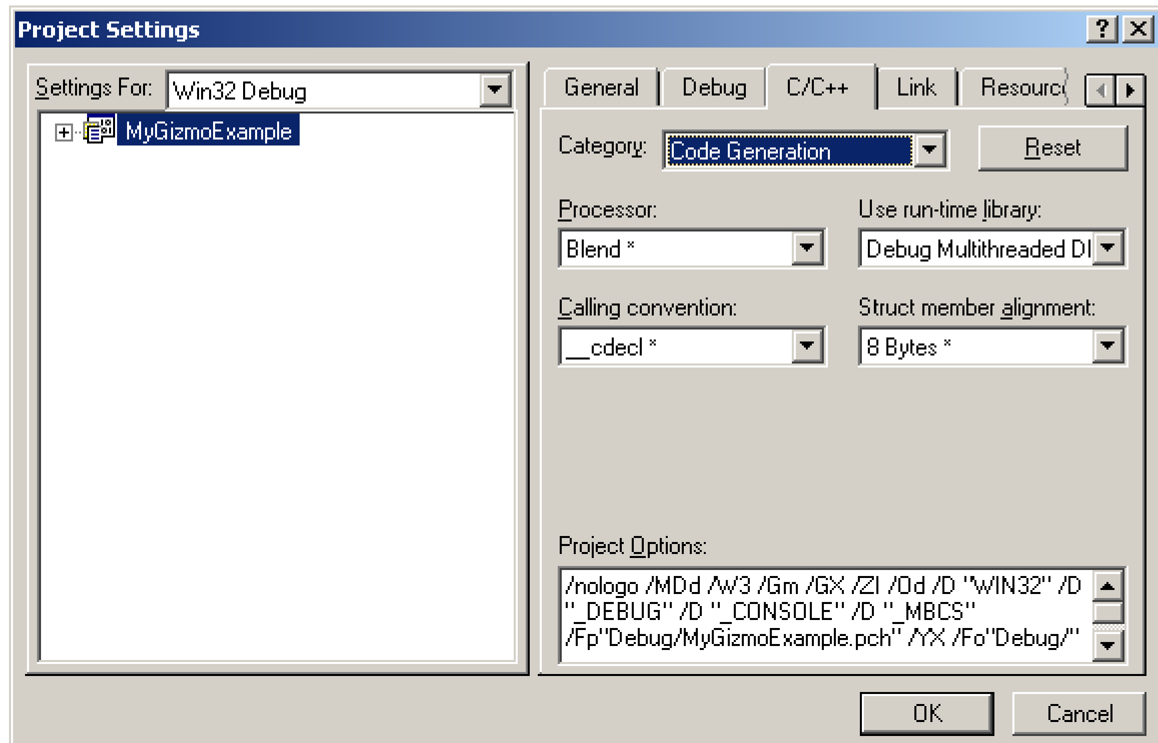
17. Select the **"C/C++"** tab.

18. Select **"Win32 Debug"** in the "Settings For" combo box.

19. Select **"Code Generation"** in the "Category" combo box.

20. Select **"Debug Multithreaded DLL"** in the "Use run-rime library" combo box.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se



21. If you want to run a release configuration, select **"Win32 Release"** in the "Settings For" combo box and then **"Multithreaded DLL"** in the "Use run-rime library" combo box.

22. Select **"All Configurations"** in the "Settings For" combo box.

23. Select **"Preprocessor"** in the "Category" combo box.

24. Type the path to the include directory for GizmoDistribution and GizmoBase in the "Additional include directories" edit box, separated with a comma. In our case **"C:\GizmoSDK\GizmoDistribution\Include, C:\GizmoSDK\GizmoBase\Include"**.

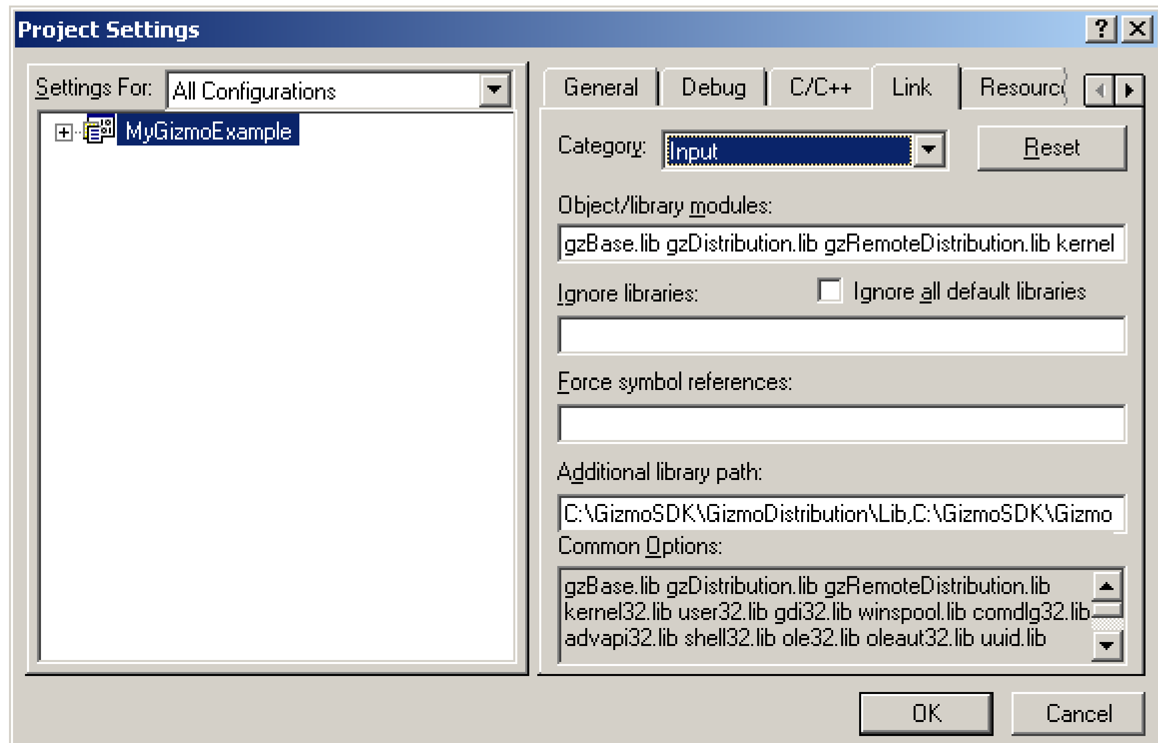
25. Select the "Link" tab.

26. Select "Input" in the "Category" combo box.

27. Add **"gzBase.lib gzDistribution.lib gzRemoteDistribution.lib"** somewhere in the "Object/library modules" edit box.
    Note: The **gzRemoteDistribution.lib** is needed only if you are using global sessions, i.e. if you need to communicate between processes or computers.

28. Type the path to the lib directory for GizmoDistribution and GizmoBase in the "Additional library path" edit box, separated with a comma. In our case **"C:\GizmoSDK\GizmoDistribution\Lib, C:\GizmoSDK\GizmoBase\Lib"**.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se



29. Press **<OK>**

30. Copy all the code from an example in the GizmoDistribution examples and paste it into your "**MyGizmoExample.cpp**" file.

31. Go to the **Build** menu and select **"Build MyGizmoExample.exe"** or just press **<F7>** to build the example.

32. Press **<F5>** to run the application! Make sure that the location of the Gizmo libraries (DLLs) are in your path.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

5 FAQ

5.1 Object and event issues

- *Does GizmoDistribution use IDL for defining types and interfaces of objects and events?*

  No. Both objects and events have dynamic attribute interfaces, i.e. they are built up of a set named attributes that are allocated in run-time. The type of an attribute is also dynamic, i.e. it could be of any type (almost), but typically a number or a string. The difference between the attributes of an event and an object is that an object can be modified, i.e. attributes can be added, removed or updated at any time. It is possible to create custom object and event types though. To do this you inherit the `gzDistObject` or `gzDistEvent` classes and implement the GizmoBase run-time type interface.

- *Why are interfaces defined in run-time instead of compile-time as for most other distributed object SDKs?*

  The purpose is to enhance the flexibility of systems. It is very easy to e.g. add a new attribute to an object or an event. You don't have to specify nor its name or its type in any IDL definition. Backward compatibility is easy to fulfill since existing don't even have to be re compiled.

5.2 Client issues

- *Are new objects notified to clients in the same consecutive order as they are added to the session?*

  Yes. Objects are added to the session and notified to clients as the requests to add objects are processed. On a local session the requests are processed by the local manager, and on a global session the requests are processed by the current global session server. This means that all clients will get notified with new objects on a session in equal order, no matter where they are located. The same applies for updated objects and removed objects.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

- *Are events notified to clients in the same consecutive order as they are sent on the session?*

  Not necessarily. Within the local process events are notified in the same order as they are processed by the manager. But on a global session events are distributed on the network directly from the process instance where the sending client is located, i.e. the events are not processed by the current global session server. This means that all clients within a single a process will get notified in the same order, but the order may be different from other processes.

- *The attributes in the notification data and the attributes in the corresponding object are not always equal. Why?*

  This is by design. The purpose of the notification data is to describe which attributes where added, updated or removed, and what the resulting attribute values were. Since notifications are asynchronous the corresponding object may have been updated again before the current update was notified.

5.3 Network issues

- *Which of the UDP transport types is to prefer? Multicast or broadcast?*

  They both have their advantages and drawbacks. We have encountered problems with multicast on some networks. On some networks broadcast is not even allowed. Ask your local network administrator for advice.

- *Is it possible to distribute data only between processes on the local host, but not to other hosts on the network?*

  Yes. Use local named pipe transports or try binding UDP multicast or broadcast transports to address 127.0.0.1 (localhost).

- *I have a client sending large bursts of events on a global session. The other clients within the local process always receive all events, but clients in other processes sometimes seem to lose events. Why? What is the solution?*

  You are probably running your channels in best-effort transfer mode. When using best-effort mode, there is no mechanism for detecting and recovering messages that are lost on the network. Try using the reliable transfer mode instead by applying an appropriate buffer size to the channel constructor. Note that all processes must use the same transfer mode.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

5.4 General issues

- *Does GizmoDistribution include any mechanism for remote method invocation (RPC or similar)?*

  No. Not yet anyway…

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se

6 LINKS

http://www.gizmosdk.com	The GizmoSDK web site.
http://www.trolltech.com/	The QT / Trolltech web site.

GizmoSDK Copyright © 2004 – Saab Training Systems AB
http://www.saabtrainingsystems.com/,
email mailto:gizmosdk_info@sts.saab.se