LiveDist user-guide

How to use the LiveDist product

17.07.2007

1 Preface

The purpose of this user-guide is to describe how to use the LiveDist middleware-product, to selectively distribute a database to many clients, in a dynamically changing topology, and with a close to real-time performance.

- The information in this guide applies to the C# / SQL-Server version of the LiveDist product (C# / Oracle and Java / Oracle versions of the product are also available).
- The LiveDist product can be used for distributing entities to:
    - End-users (displaying data using browsers or fat-clients)
    - External systems
    - A topology of inter-connected sites, where each site functions both as a client of the sites it is directly connected to, and as the distribution-server of these sites. All sites are allowed to concurrently modify entities, and there is no master-site.

1.1 Information in this guide

As a software developer, you should learn about internal LiveDist features, which can be configured to improve performance, scalability, and availability. This guide describes LiveDist features that relate to application-framework development. It does not cover the internals or deployment options of the product.

1.2 Audience

The users-guide is intended for programmers developing the entities sub-system, which wraps the LiveDist product and hides it from the application.

This guide assumes that you are familiar with:
HTTP 1.1, ASP.NET 2.0, ADO.NET 2.0, and C# 2.0.

1.3 XML documentation in the source-code

To ease maintenance, details which can be found in the C# source-code (expressed as XML Documentation comments), are not duplicated in this manual. Instead, links to these pieces of information are provided.

1.4 Our web site

For additional information please visit our web-site: http://www.galiel314.com or http://www.livedist.com

1.5 How this guide is organized

This user-guide contains the following major parts

1.5.1 Introduction

This part presents a high-level introduction to dynamically-selective distribution, and defines the notions used throughout the rest of this manual

1.5.2 Entities

This part describes the distributed entities.

1.5.3 Delta-objects

This part describes how logs of database-modifications are represented as C# objects. The distribution-schema is also described.

1.5.4 Plugs

This part describes all LiveDist plugs. A plug-implementation is an application-supplied code, which implements a specific plug-interface, and invoked by LiveDist components.

1.5.5 Deployment

This part describes how to deploy LiveDist servers to **IIS** (Microsoft's web-server, which is part of the windows operating-system), and to **Cassini** sample web-server (an open-source freeware project from www.asp.net).

2 Introduction

2.1 Dynamic selective distribution

2.1.1 Topics

LiveDist supports dynamic selective distribution. What is selective distribution? The meaning of selective distribution is that each entity in the database has a list of distribution-attributes. Each distribution-attribute is actually a string, and is called a **topic**. Topic values are not predefined, and they are generated dynamically by the application.

The meaning of each topic value is a convention between the application running on the server-side and the application running on the client-side. The application should define a naming-convention for topic-values. A single application-software -component should manage the allocation of topic-values.

2.1.1.1 Topics-Lists

A comma-separated list of topics is called a **topics-list**. The topics-list of each entity, is stored in a varchar column, in the database row in which the entity is stored. Topics-list values are, generated dynamically by the application, when inserting new entities into the database. The topics-list of an existing entity may be modified by the application, when updating the entity. Note that in the topics-list ",A1,A2,B1," a comma precedes the first topic, and a comma follows the last topic. The reason for that is to enable the selection of entities having a given topic, by the SQL statement:

```
SELECT * FROM Xxx WHERE Topics LIKE '%,A1,%'.
```

Note: When a topic (in a topics-list) starts with the character '**#**', the Delta-server ignores the '#' prefix, and the Cache-server ignores the entire topic. These topics are used for preventing loop-back distributions (sending a delta back to the server from which the delta has been received).

2.1.1.2 TopicsSlices

Topics-slices are selection criterias for entities, based on the existance of topics, in the topics-list attached to each entity (and on the value of the UpdateTime field). A **topics-slice** is an expression containing topic-values (and time-windows), connected via parentheses and operators:

```
TopicsSlice ::= Topic |
          TopicsSlice * !Topic |
         (TopicsSlice) |
          TopicSlice + TopicSlice |
          TopicSlice * TopicSlice |
          TopicsSlice * TimeWindow999 |
          TopicsSlice * TimeWindow[dd/mm/yyyy hh:mm:ss]
* ::= and-operator
+ ::= or-operator
! ::= not-operator
999 ::= number of seconds in the time-window
            ( now - nnn <= UpdateTime <= now)
[dd/mm/yyyy hh:mm:ss] ::= defines an absolute time
         (example: 31/12/2005 14:59:00)
```

The characters '*****', '**+**', '**!**', '**(**', '**)**', '**,**','**#**' are not allowed in topic-values.

Example: the topics slice (T2+T3*(T4+T5))*!T1 selects all entities for whom topics-lists don't contain the topic T1 but contain T2 or T3 and (T4 or T5).

Each distribution client subscribes to a **topics-slice**, and receives all entities that are selected by the client's subscription. Each distribution client additionally receives entities on which it is registered (all entities known to exist in the client's database). As an exception to the above rule, a !T in the client's topic-slices prevents the selection of delta-objects with T in their topic-list, even when the client is registered on the entity.

Note: The terms in the topics-slice are processed left-to-right. All operators have same precedence, and parentheses may be used to change order.

Note: Time-windows are either relative (defines how many seconds in the past it contains), or absolute (contains any time greater or equal than the specified time)

2.1.2 Time-window

Some clients are not interested in entities which have not been updated for a long time. These clients can include time-window conditions in their topics-slices. For example, the topics-slice "(T1+T2)*TimeWindow3600+(T3*TimeWindow120)" selects all entities having topic T1 or T2 that have been updated (or created) in the last 60 minutes, and also selects all entities having topic T3 that have been updated (or created) in the last 2 minutes. Time-windows are only used when loading from the CacheServer, and they are disregarded in steady-state.

2.1.3 Entity→Client registration

Once an entity is distributed to a client (the entity has been selected by the client's topics-slice, during a load or during steady-state distribution), it is registered as such in the LiveDist server, and the client will receive updates on that entity from the server, even when the client's topics-slice no-longer selects the entity.

As an exception to the above rule, when the client's topics-slice contains a !T term (which means that the client chooses not to receive delta-objects containing the topic T), the client will not receive delta-objects having T in their topics-list, even when the client is registered on the entities of these delta-objects.

The client may explicitly request the server to un-register specific entities, when it decides that it should no longer receive them.

The registration mechanism is used to enable correct propagation of updates between server-sites, when there is no master-site. Note that it is possible that, due to unsynchronized concurrent updates of the topics-list of entities in multiple sites, the topics-list of entities may be temporarily inconsistent between server-sites.

2.1.3.1 Registered versus unregistered clients

When a site uses an unregistered Edge instance to connect to another site (the value of the constructor argument useRegistration is false), no registration is used for that client, and the client is assumed to be synchronized on all entities, which are selected to be distributed to the client by its topics-slice (subscription). Thus, partial-delta-objects (and not full-delta-objects) are always sent to the client.

A registered client (Edge) receives from its server, all entities selected by the client's topics-slice (subscription), and all entities known to the server as existing in the client's database (registration). After the server has passed a full-delta to the client, the client is registered-as-synchronized on the entity. The client may inform the server that it is no longer interested in receiving specific entities (un-register).

2.1.4 Dynamically changing topology and subscription

LiveDist supports a dynamically changing topology. What is a dynamically changing topology? The clients of each distributed-database are neither statically defined, nor require administration. Clients can join or leave the network at any time.

Each client uses a software component, called **edge**, to interact with LiveDist servers. The edge uses the HTTP 1.1 protocol, for talking with LiveDist servers.

2.2 New clients

When a client joins the network for the first time, it calls the method
edge.FullLoad(topicsSlice**)**.

The client passes its requested topics-slice to the edge, and receives a **full-load** (create-delta-objects, with values for all not-null fields) of all selected-entities. The topicsSlice is the new subscription of the client. The client is registered-as-synchronized on all returned entities. When an entity which has been fully returned in a load, is updated again, a partial delta-object will be returned (because the entity is synchronized between the two sites).

A global (per site) numeric **sequence-number** (of type long) is attached to each transaction, which modifies the database. Sequence-numbers are greater for more recently committed transactions. Thus, sequence-numbers can be used for representing discrete points in time.

A sequence-number is included in each snapshot received by a client. It represents the point of time when the snapshot has been taken. After the client finishes processing the loaded entities, it starts receiving distribution-updates beginning from that sequence-number.

Note: New clients may also receive their first load by invoking the service edge.SyncLoad (see below).

2.3 Change of subscription

When a client in steady-state needs to change its subscription, it calls the method:

```
                    edge1.PartialLoad(newTopicsSlice).
```

The client passes its new topics-slice to the edge, and receives a **partial-load** of all entities that are newly selected by the client's subscription (selected by newTopicsSlice, and not yet registered to the client). The client is registered-as-synchronized on all returned entities.

The client is required to delete from its local database all the entities which are no longer selected by newTopicsSlice. For this purpose, the client uses another edge object (edge2), which is connected to the local LiveDist server. The client receives from that edge, a list of all entities existing in the local database but no longer subscribed to this client, by calling:

```
                    TypedIdList[] minus =
                            edge2.SelectMinus(oldTopicsSlice, newTopicsSlice).
```

The client then deletes these entities from the local database,
and then calls:

```
                    edge1.UnregisterEntities(removedGuids, loadSequence).
                            // the same sequence applies to all entities
```

edge1: An edge connected to LiveDist servers at remote-site.
edge2: An edge connected to LiveDist servers at local-site.

Note: When creating/updating an entity in the current site's database, the entities sub-system calculates the value of the Topics fields. Thus, the topics-list of each entity in the local Cache-Server are correct, even after receiving a delta-object from another site.

2.4 Synchronization

When site X reconnects to site Y following a communication failure which is too lengthy, or when site X connects to site Y for the first time, it must synchronize itself with site Y (site Y will also synchronize itself with site X). This is done by calling the method:

```
// Following a long communication failure:
edge.SyncLoad( // without registration
        topicsSlice,
        lastTransmissionTime).
   or
// When connecting to another-site for the first time:
edge.SyncLoad( // with registration before load
        iterOfGuids,
        topicsSlice,
        lastTransmissionTime).
```

The client passes its current topics-slice, and the last-transmission-time to the edge and receives a **sync-load**.

The last-transmission-time is the last time that the client received any steady-state update-message from the server, minus a few-minutes (to compensate for the clock differences between the server and the client). The client passes DateTime.MinTime (or an application-dependent old enough time in the past) as UpdateTime, when it changes its server.

All entities, which are selected by the topicsSlice (or by registration) and which were modified (or created) after the last-transmission-time, are fully loaded to the client. Each client is registered-as-synchronized on all returned entities.

Note: when an entity should be deleted, the entity is marked by the application as logically-deleted. The actual deletion of entities from the database is separately done in each site as a maintenance operation. Thus, physical deletion of entities is irrelevant to synchronization.

When a client that is not yet registered at the server invokes edge.SyncLoad (the without registration overload), exception NotRegisteredException is thrown, and the client must register by invoking the service:

edge.RegisterEntities(iterOfGuids),

to tell the LiveDist servers on which entities it is currently registered. The edge will be registered-as-synchronized on all specified entities. The above mentioned exception may occur, for example, when all LiveDist servers in the site have rebooted.

The client may invoke the service SyncLoad (the with-registration overload), to both register and synchronize in the same service-invocation.

2.5 Steady-state distribution

A client is in steady-state mode, when it is connected to LiveDist servers and is receiving binary **delta-buffers,** containing logs of database updates made on all subscribed or registered entities. This is done by calling the method:

edge.FetchDeltas().

This method returns a log of all database-updates, made on subscribed (or registered) entities in one or more transactions. The sequence-number of the last returned transaction is attached to the received delta-buffer. The client can read the current sequence-number of the edge, by getting the value of the **Sequence** property. FetchDeltas waits (blocks the current thread) until new delta-buffers arrive.

If the client is not yet registered at the server, exception NotRegisteredException is thrown, and the client must register by invoking the service:

edge.RegisterEntities(iterOfGuids**)**,

to tell the LiveDist servers on which entities it is registered, and than recall FetchDeltas.

When a client receives updates for entities in which it is no longer interested, the client invokes the service:

edge.UnregisterEntities(guids, lastReceivedSequence**).**

After a temporary failure, such as a communication failure, a client in steady-state simply re-invokes the service edge.FetchDeltas.

When a client in steady-state reboots, it calls:

edge.SkipLoad(currentTopicsSlice, lastReceivedSequence**)**,

and then it invokes the service FetchDeltas.

Note: Logical-deletion of an entity is done by updating a field to a value which marks the logical-deletion. Thus, logical-deletions of entities are distributed as update operations. Physical-deletions of entities are separately done in each site as maintenance operations. For each deleted entity, a delta-object with delete operation should be created, for telling LiveDist cache-servers to remove the entity from their in-memory-database. Clients of LiveDist should ignore these delta-objects.

In steady-state, the entities sub-system is responsible for creating the delta-objects at the end of each transaction, and to pass these delta-objects to LiveDist, by calling the Append method.
When loading entities from a Cache-server to any client, the Cache-Server is responsible for creating the create-delta-objects, which are returned to the client.

2.6 Father-sons topology

In a father-sons topology, we mean that when an entity exists in the database of a site, it must also exist in the database of its father-site. Each site has at most one father-site, and each father-site can have many son-sites (no circles are permitted). Thus, each site sees a portion of the database of its father-site.

Each son-site, receives from its father-site, all entities selected by the son-site's topics-slice (subscription), and all entities known to the father-site as existing in the son-site (registration). After the father-site has passed a full-delta to the son-site the son-site is registered-as-synchronized on the entity. The son-site may inform the father-site, that it is no longer interested in receiving specific entities (un-register).

Father-sites use unregistered Edge instances to connect to their son-sites.
Son-sites use registered Edge instances to connect to their father-sites.

The father-sons topology is just an example of a possible topology.
The LiveDist product can be used for supporting other topologies as well.

2.6.1 Routing local updates to father and sons

When a new entity is created in a site, a create-delta-object is sent to the father-site (always), and to all son-sites selecting the entity via their subscriptions.

In the **create-delta-object**:

UpdateTime = field UpdateTime in entity
OldTopics = null
Topics = father-topic, application-topic1, application-topic2, …
DeltaOper = CREATE
Values of **all** fields

By father-topic, we mean a constant-topic (not specific to a specific site), representing (in any-site) the father-site of the current site.

When an existing entity is updated in a site, a **partial-delta-object** and a **full-delta-object** are created (in that order, so that son-sites registered-as-unsynchronized on the entity, will not receive the partial-delta-object after receiving the full-delta-object).

In the **partial-delta-object**:

UpdateTime = field UpdateTime in entity
OldTopics = previous value of Topics
Topics = father-topic, application-topic1, application-topic2, …
DeltaOper = PARTIAL
Values of **modified** fields

In the **full-delta-object**

UpdateTime = field UpdateTime in entity
OldTopics = previous value of Topics
Topics = application-topic1, application-topic2, …
DeltaOper = FULL
Values of **all** fields

A partial-delta-object is a delta-object, in which the DeltaOper is PARTIAL.
A full-delta-object is a delta-object, in which the DeltaOper is FULL.
Note: the FULL and PARTIAL delta-operations are only used in first-class delta-objects. In second-class delta-objects we use the operations CREATE, UPDATE, or DELETE.
All entities are distributed to the father-site. Thus, the father-site is assumed as implicitly registered-as-synchronized on all entities in the son-site.

Performance improvement:
When an existing entity is updated, and its topics-field is not modified, and the entity has recently been updated, the above delta-objects can be replaced by a single **update-delta-object**, which is similar to the partial-delta-object, but DeltaOper = UPDATE. This will bypass the need for generating the full-delta-object. The improvement is based on the fact that a client may need the full-delta-object, only when the entity enters the client's slice due to a topic-change, or when it sees the entity for the first time because the entity has not been updated for a long time before this update.

Entities are distributed to son-sites according to their subscriptions and registrations:

A son-site will receive the full-delta-object (of an updated entity):

- ♦ When it sees the updated entity for the first time
- ♦ When the son site is registered-as-unsynchronized on the entity, meaning that the entity is known to exist in the son-site, but possibly the son-site has not yet received a full-delta-object from the father-site (the son-site will receive the full-delta-object, and will also be registered-as-synchronized on the entity).

A son-site will receive the partial delta-object, when the son-site is registered-as-synchronized on the entity.

2.6.2 Receiving creations and updates from the father site

When a create-delta is received from the father site, a create-delta-object is sent to all son-sites selecting the entity via their subscriptions.

In the **create-delta-object**:

UpdateTime = field UpdateTime in entity
OldTopics = null
Topics = application-topic1, application-topic2, …
DeltaOper = CREATE
Values of **all** fields

The father-topic is not listed in the Topics field, thus the create-delta-object is not distributed back to the father-site.

When an update to existing entity is received from the father site and rejected, then nothing is distributed, because the last update of the entity in local database has been or is being distributed to the father site (all updates are) and will be accepted there. Also, nothing is distributed to the son-sites, because the local database has not been modified.

When an update to existing entity is received from the father-site and accepted, a **partial-delta-object** and a **full-delta-object** are created by the entities sub-system (in that order).

In the **partial-delta-object**:

> UpdateTime = field UpdateTime in entity
> OldTopics = previous value of Topics
> Topics = application-topic1, application-topic2, …
> DeltaOper = PARTIAL
> Values of **modified** fields

In the **full-delta-object**

> UpdateTime = field UpdateTime in entity
> OldTopics = previous value of Topics
> Topics = application-topic1, application-topic2, …
> DeltaOper = FULL
> Values of **all** fields

The father-topic is not listed in the Topics field, thus the partial-delta-object and the full-delta-object are not distributed back to the father-site.

The distribution to son-sites is similar to distribution of updates originated in the local-site.

The same performance-improvement, of replacing the 2 delta-objects by a single update-delta-object can be used (as described above).

2.6.3 Receiving creations and updates from son-sites

When a create-delta-object is received from a son-site, a create-delta-object is created by the entities sub-system and sent to the father-site and to all other son-sites selecting the entity via their subscriptions.

In the **create-delta-object**:

UpdateTime = field UpdateTime in entity
OldTopics = null
Topics = father-site, not-son-site, application-topic1, …
DeltaOper = CREATE
Values of **all** fields

Two site-specific topics are allocated to each son site:

- The direct-naming topic **son-site** (for specifying that a delta-object should be sent to the site)
- The negation-topic **not-son-site** (for specifying that a delta-object should not be sent to the site). Each son site combines (in its subscription), the not-son-site topic with a not-operation, for preventing ping-pong distribution.

When an update to existing entity is received from a son-site and **rejected**, then a register-delta-object, and a full-delta-object are created (in that order), to register the son-site on the entity and to distribute a full-delta-object to the son-site (when the son-site is not yet registered-as-synchronized on the entity). Nothing is distributed to the father-site and to other son-sites, because the local database has not been modified.

In the **register-delta-object**:

Topics = son-site
DeltaOper = REGISTER

In the **full-delta-object**:

UpdateTime = field UpdateTime in entity
OldTopics = null
Topics = application-topic1, application-topic2, …
DeltaOper = FULL
Values of **all** fields

When the son site is registered-as-unsynchronized on the entity (meaning that the entity exists in the son-site, but possibly the son-site has not yet received a full-delta-object from the father-site), the son-site will receive the full-delta-object, and the son-site will also be registered-as-synchronized on the entity. Other son sites will receive the full-delta-object only when they are registered-as-unsynchronized on the entity.

When an update to existing entity is received from a son-site and accepted or partly accepted, a **register-delta-object**, a **partial-delta-object** and a **full-delta-object,** are created by the entities sub-system (in that order).

In the **register-delta-object**:

- Topics = son-site
- DeltaOper = REGISTER

In the **partial-delta-object**:

- UpdateTime = field UpdateTime in entity
- OldTopics = previous value of Topics
- Topics = father-site, not-son-site, application-topic1, …
- DeltaOper = PARTIAL
- Values of **modified** fields

In the **full-delta-object**

- UpdateTime = field UpdateTime in entity
- OldTopics = previous value of Topics
- Topics = application-topic1, application-topic2, …
- DeltaOper = FULL
- Values of **all** fields

The father-site will receive the partial-delta-object (the father-site is always synchronized on all entities).

The register-delta-object causes the received-from son-site to be registered-as-unsynchronized on the entity (only when not yet registered on the entity). The received-from son-site, will not receive back the partial-delta (the not-son-site topic prevents that). The received-from son-site, will receive the full-delta-object (and be registered-as-synchronized) if not yet registered-as-synchronized on the entity.

The distribution to other son-sites is similar to distribution of updates originated in the local-site.

Note that the difference between accepting an update from a son-site and rejecting the update is that when fully rejecting the update the partial-delta-object is not created, because the local database is not updated. In both cases all son-sites which are unsynchronized on the entity will receive the full-delta-object.

2.7 Controlling client servicing

The database table LiveDistClients is used by application
for managing the clients of the current site.

The table is defined as follows:

```
    LiveDistClients (
            ClientId varchar(64) not null unique,
            BitNumber smallint not null unique
                    check(0<=BitNumber and BitNumber<64),
            IsDisabled bit not null)
```

Triggers which are defined in this table prevent updating the ClientId and BitNumber columns of existing rows. The application may dynamically add and delete rows. The application may also dynamically update the IsDisabled column of any row.

LiveDist servers listen to changes in this table (but never update it themselves).

For each client a row should be inserted into the table.

The ClientId is the unique name of the client, passed to the instance of the implementation of IHttpClient, which has been passed to the constructor of the Edge instance.

BitNumber is the bit in the 64-bits client’s mask allocated for that client (in the range: 0 to 63).

When the value of IsDisabled is updated to 1, LiveDist servers abort all active requests invoked by the disabled client, and refuse to service new requests from that client. When the value of IsDisabled is updated to 0, LiveDist servers serve requests invoked by the enabled client.

3 Entities

3.1 Entities hierarchies

The LiveDist product distributes persistent entities which are stored in the database. Each **entity** is a hierarchy of objects (objects-tree). When stored in a relational-database, a single entity may span several database rows, at least one row for the entity and one row per each collection-member.

Entities are sometimes called **first-class** or **top-level** entities. Sub-entities, which can only exist as part of other first-class-entities or other sub-entities, are sometimes called **second-class** entities.

3.2 The entities sub-system

The **entities sub-system**, is responsible for handling the object/relational mapping of entities, and to track changes in entities. The LiveDist product can't directly fetch entities from the database, because the object/relational mapping is controlled by the entities sub-system. Thus, LiveDist components request the entities sub-system to fetch entities from the database for them, by using a configurable callback mechanism called plugs.

3.3 Tech-ids

Each entity has a globally unique identifier, called **tech-id,** of the type Guid. Once created, the tech-id of an entity can't be modified. The tech-id is used as the primary-key of the row in which the entity is stored. When an entity needs to reference another entity, it defines a link-field (of type Guid) containing the tech-id of the referenced entity. To preserve integrity, the database column which stores the value of a link-field should be defined as a foreign-key,

3.4 Topics-lists

Each entity has a **topics-list** field, enabling the application to add/remove distribution-topics to/from the entity.

3.5 UpdateTime

Each entity has an "UpdateTime" field (of the type DateTime). It contains the last-time when the entity was created or updated.

3.6 Loading into cache

Each entity has an **InCache** field; used for controlling whether the entity is fully-loaded into the memory of the LiveDist cache-servers. This field is assigned a value when the entity is created, and can't be modified afterward. When an entity is not fully-loaded into cache, only its header-fields are loaded (TechId, Topics, and UpdateTime).

4 Delta objects

4.1 Passing to Append method

Before the end of each transaction which modifies the database, the entities sub-system creates at least one **delta-object** per each modified entity and one delta-object per each modified sub-entity, and passes the collection of the delta-objects, corresponding to all modified first-class entities, to the static method:

For distributed transactions:

LiveDist.Api.Reception.Append(sqlconToDB, deltas**)**

Or

For local SQL-server transactions:

LiveDist.Api.Reception.Append(sqltransToDB, deltas**)**

Each delta-object is an instance of a **delta-class**. Delta-classes are used for representing modifications on entities/sub-entities as C# objects.

After the transaction is successfully committed, the entities sub-system may call:

LiveDist.Api.Reception.NotifyDeltaServer().

This method sends a multicast message to the master delta-server, which responds by fetching the new delta-buffers from the log-table in the database, inserting the new delta-buffers into its in-memory data-structures, and then pushing delta-objects to its clients (according to the clients subscriptions and registrations). Thus the effect of calling the method NotifyDeltaServer is the instant distribution of new data, with no delay, to all edges.

The entities sub-system may skip calling NotifyDeltaServer. When the master delta-server doesn't receive a multicast message for some time, it periodically fetches new delta-buffers from the log-table, and pushes the new delta-objects to its clients.

4.2 Receiving from Edge methods

Delta-objects are returned to distribution-clients by instances of the class **LiveDist.Api.Edge**, both when loading entities, and when fetching steady-state distribution-updates. A delta-object returned during a load is called a **create-delta**, in which all non-null fields have values.

4.3 Exact definition

For an exact definition of the structure of delta-classes, read the source-code documentation for the LiveDist.Api.Reception class.

4.4 The distribution schema

The source-code of the delta-classes, is generated from the data-dictionary, in a mechanism that is external to the LiveDist product. The project's development-environment should generate an XML-file, in which all delta-classes are listed. This file is passed as an argument to the utility program **MakeSchema**. This program learns the structure of the listed delta-classes by using reflection. MakeSchema saves all the required information about the entities/sub-entities and their fields in a formatted string called **schema**, and stores the new schema-definition and its **schema-name** in the database. When the schema is changed, its attached schema-name must also be changed.

The schema is used by LiveDist components for encoding/decoding delta-objects into/from binary **delta-buffers** (transmittable messages), by using reflection.

The **schema-name** is stored as a header-field in each transmitted delta-buffer. This enables the detection of data-dictionary changes on the fly. The schema-name should be a short string containing a version-number.

The format of the XML-file is as follows:

```
<LiveDist>
    <DeltaClass className="x1.x2.xx,a" />
    <DeltaClass className="y1.y2.yy" />
     ...
</LiveDist>
```

className: is a full class-name, including its containing namespace, and the name of the assembly in which it is deployed. In the example above, the class "xx" is contained in the namespace "x1.x2", and is deployed in assembly "a" (the file a.dll). When no assembly is specified, the default assembly is used (the value of the constant LiveDist.Constants.DEFAULT_ASSEMBLY).

When the delta-class A is a base-class of the delta-class B, then A must be listed before B in the XML-file.

5 Plugs

All the interfaces of LiveDist plugs are defined in the **namespace**:
LiveDist.Plugs

A detailed description about each plug, can be found in the C# source-code, of the plug's interface.

5.1 IHttpClient

Used by edge instances for invoking HTTP-services in LiveDist servers, and by LiveDist servers for invoking HTTP-services in other LiveDist servers. The implementation of this plug controls the parameters-passing mechanism to/from HTTP-services (parameters can be passed as part of the URL, as header-fields, or in the body of the request/response). A reference implementation for this plug is supplied with the product.

5.2 IHttpServer

Used by LiveDist servers for analyzing http-requests, retreiving input args, and for returning responses to the clients. The arguments-passing mechanism must match the mechanism that is used by the implementation of IHttpClient. A reference implementation for this plug is supplied with the product.

5.3 IDeltaObjectsReader

When a LiveDist cache-server needs to fetch entities from the database, it uses an implementation of this plug to do so. The implementation uses the entities sub-system for accessing the database.

5.4 ICacheNotifier

This plug is used by cache-servers to notify the entities sub-system
that the topics-slice (subscription) of an edge-instance has changed.
This plug is also used by cache-servers to notify the beginning and ending of snapshots to the entities sub-system.

5.5 IControlAndParams

Used by LiveDist components for:

- Getting the values of config-parameters from the application
  (an alternative to using the web-config file).
- Reporting changes in the status of sub-systems to the application.
- Reporting to the application when entering or exiting exceptional conditions.
- Reporting special events that don't cause a status change to the application.
- Assigning values to statistical-counters.
- Checking if an active snapshot should be aborted.

A reference implementation for this plug is supplied with the product.

5.6 IPermit

Used by the delta and cache servers for verifying that the client is authorized to request the given topics-slice and classification.
A reference implementation for this plug is supplied with the product.

6 Limitations

6.1 The number of direct son-sites of any site

The current implementation of the registration-table enables each site to have up to 64 direct sons of its own. Of course, each son-site may have up-to 64 direct sons, and so on.

7 Deployment

7.1 The LiveDist web-application

The LiveDist server-side components (delta-server and cache-server) are packaged together as a web-application, hosted by a web-server (which is capable of hosting the ASP.NET 2.0 runtime). The LiveDist web-application can be configured to boot the DeltaServer, the CacheServer, or both. Thus, each machine can be configured to run a single LiveDist server (delta or cache) or to run both servers (as two http-handlers in the same web-application).

The LiveDist web-application is entirely configured by entries in the **web.config** file.

An example of a web.config file:

```
<?xml version="1.0"?>
<configuration>
   <system.web>
       <httpHandlers>
           <add verb="*" path="LiveDist*/Delta.config"
                type="LiveDist.Core.HttpDeltaHandler,LiveDist"/>
           <add verb="*" path="LiveDist*/Cache.config"
                type="LiveDist.Core.HttpCacheHandler,LiveDist"/>
       </httpHandlers>
       <compilation debug="true"/></system.web>
       <appSettings>
         <add key="LiveDistConnectionString" value="Server=.../ ">
         <add key="LiveDistUseRegistration"  value="true"/>
         <add key="LiveDistStartDeltaServer" value="yes"/>
         <add key="LiveDistStartCacheServer" value="yes"/>
       </appSettings>
</configuration>
```

In the above web.config file, the following configuration parameters are defined:

- **LiveDistConnectionString**: the connection-string to the SQL-server database. Whenever LiveDist creates an instance of SqlConnection, it passes this string to its constructor.
- **LiveDistUseRegistration**: when “true”, a distributed registration-table is kept in the memory of all LiveDist servers in the site, and registration is used for Edge instances constructed with userRegistration == true.
- **LiveDistStartDeltaServer**: when “true”, the LiveDist web-application boots the DeltaServer, when started by the web-server.
- **LiveDistStartCacheServer**: when “true”, the LiveDist web-application boots the CacheServer, when started by the web-server.

Note: The IIS web-server starts the LiveDist web-application, when a HTTP service that is routed to the DeltaServer or the CacheServer (because of its URL) is invoked for the first-time.

7.2 LiveDist instances

In each server machine, few instances of LiveDist can be deployed, as different web-applications (each configured by its own web.config file). LiveDist several instances are unaware of each other. They distribute different databases or different tables in the same database. LiveDist instances are deployed as different web-applications in the same web-site, or as different web-sites in the same machine. Either way, they run in different application-domains.

For each instance the following definitions in the web.config file are required:

> The `<httpHandlers>` section in the web.config file should define the paths to the delta and cache servers. For example, the **path** to the DeltaServer of LiveDist instance number 2 is: “LiveDist**2**/Delta.config”. These definitions instruct the web-server to route all services, with the declared paths, to LiveDist http-handlers.
>
> The value of the parameter **LiveDistApplicationSuffix** is used, by the class PlugimpI.HttpClient, as the XXX in the URLs “LiveDistXXX/Delta.config” and “LiveDistXXX/Cache.config”. The default is “1” (stands for instance 1).
>
> The value of the parameter **LiveDistTablesPrefix** is used as the prefix of all LiveDist tables in the database. The default value is “**LiveDist..**”, which means that all LiveDist tables are defined in the database named LiveDist.

For each instance, the supporting LiveDist database tables, for that instance, must be created in the correct database. This is performed by running the script **LiveDistDB.sql**, after renaming all references to database LiveDist to the database name used by the configuration parameter LiveDistTablesPrefix (and/or after renaming all table-names according to the table-name-prefix defined in the configuration parameter LiveDistTablesPrefix).

For each instance, a directory or a site must be added to **IIS**. When adding a directory or a site, we define the path to the directory containing the LiveDist web-application. The web.config file, in that directory, is used for configuring the LiveDist instance.

When using **Cassini**, an instance of the light-weight Cassini-web-server is executed, per each LiveDist instance. The used port-number and the path to the web-application’s directory, are passed to the Cassini web-server (as command-line arguments, or as textual-fields to its GUI).

7.3 URLs and HTTP-methods

The URLs, used for invoking services of LiveDist servers, are determined by the implementation of the IHttpClient plug. The reference implementation of this plug (Plugimpl.HttpClient), uses the following URLs for invoking services of LiveDist servers:

- http://node:[port]/LiveDistXXX/Delta.config (for DeltaServer services),
- http://node:[port]/LiveDist XXX/Cache.config (for CacheServer services).

node ::= node-name or IP-address of the machine
port ::= optional port-number (8080 when not specified)
XXX ::= the instance of LiveDist ("1", "2", …)
The ".config" file-type, tells the web-server to use ASP.NET.

The class Plugimpl.HttpClient uses the HTTP-method "POST", for invoking LiveDist services, because input arguments are passed to the services.

You can use the above URLs, for verifying that the servers are up and running. For example, specify the address: http://localhost:8080/LiveDist1/Delta.config to the browser (which by default uses the HTTP-method "GET"), to check if the delta-server is up and running on the local-host at port 8080.

8 Proxy servers

8.1 Using proxy LiveDist servers

LiveDist clients running in geographic sites other than the central-site, are able to load snapshots and receive steady-state distribution, from proxy LiveDist servers in their local-site, instead of connecting to LiveDist servers in the remote central-site. Once entities are down-loaded into a proxy LiveDist CacheServer, they can be loaded from that proxy CacheServer to all Edge instances in the same site. Steady-state distribution is transmitted from the central-site, to the master proxy DeltaServer of each site, and all Edge instances in remote sites receive steady-state distribution from their local proxy DeltaServers.

A LiveDist server is considered a proxy, when the value of the environment-variable named **LiveDistSiteName** is not **"center"**, meaning that the server is running in a site whose name is not "center".

Multiple LiveDist servers are allowed to run in the same site. They automatically load balance the load of Edge instances between them. Each Edge instance in a site, is allowed to connect to each LiveDist server in its local-site (the local-site-name is passed to the constructor of the HttpClient instance, that is passed to the constructor of the Edge instance). A proxy LiveDist server may instruct any Edge instance, to connect to another proxy LiveDist server in the local-site, for load-balancing Edge instances between proxy LiveDist servers in the site.

8.2 Status of connection to center

The LiveDist proxy servers try to keep alive active connections with the center, and they inform each Edge instance the current status of connection to center via the property **Edge.ConnectedToCenter**. This property enables LiveDist clients, in sites other than the central-site, to know when the central-site is reachable.

8.3 Selective distribution to proxies

Usually the master delta-server in each site, receives all database updates from its mother-site. Thus it can serve any FetchDeltas request, no matter which topics are used in the topics-slice. The cost of this flexibility, is that the proxy delta-server may receive updates on entities which are not selected by the topics-slice of any Edge instance in the site. This may be a problem for sites connected via small bandwidth communication.

To overcome this problem LiveDist supports selective distribution to proxies. Selective distribution to proxies is enabled when the environment variable **LiveDistDeltaUseTopicsSlice** is set to "**true**" in the web.xml configuration-file. This is allowed only when the value of the env-var LiveDistSiteName is not "center" (meaning that the current web-server is a LiveDist proxy).

When selective distribution is enabled, the master DeltaServer receives incoming distribution from its mother site, by using the FetchDeltas service instead of using the ReadDeltas service. The used topics-slice is:

fixed-topic1**+…+**fixedTopicX**+**dynamic-topic1**+…+**dynamic-topicY

The fixed-topics are listed in the env-var **LiveDistDeltaFixedTopicsInSlice**, as a comma-separated list of topics (for example: "**t1,t2,**"), that are always received from the mother-site, even when no Edge instance in the site is currently interested in them.
The dynamic-topics are non-fixed-topics used in the topics-slice of any Edge instance in the current-site. Dynamic topics remain the topics-slice of the master DeltaServer, until they are no longer in use (by any Edge instance) for a number of minutes defined in the env-var **LiveDistDeltaUnusedTopicsLifeInMinutes**.

When an Edge instance invokes FetchDeltas, with a topics-slice containing a topic not in the topics-slice that is currently used by the master DeltaServer, the request is rejected, and the client is required to invoke a load-service of the local CacheServer. The load-service will cause all topics contained in the loaded slice to be added to the topics-slice of the DeltaServer, as dynamic-topics (when not already in the slice).