ClearConnect

Concepts and Programming Manual

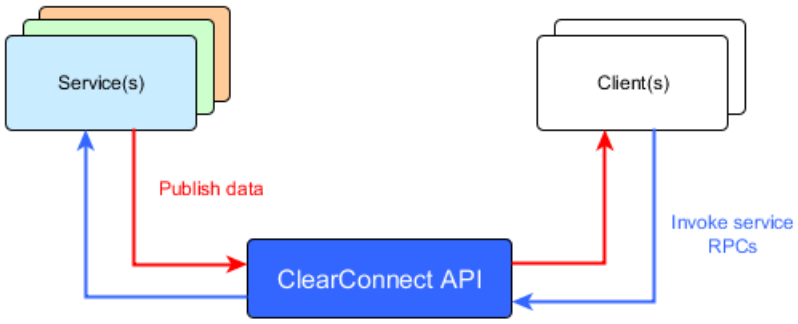
Status:	Current
Version:	0.8
API:	3.7.0 and above
Authors:	Ramon Servadei
	Paul Mackinlay
	James Lupton
Date:	10/04/2016

The ClearConnect Platform

Concepts and Programming Manual

Table of Contents

Introduction



ClearConnect is an open-source micro services platform written in Java. ClearConnect enables complex applications to be decomposed into logical “micro” services that are independently deployable, scalable and discoverable.

Applications built with ClearConnect instantly get these out-of-the-box features:

- real-time data transfer
- high-availability
- real-time monitoring and data inspection

ClearConnect makes developing distributed applications quick and easy by providing a simple application programming interface (API) and management tools.

Design Concepts

This chapter describes the core design concepts of the platform.

Data Model Concepts

All data on the platform is represented using *records*. A record is a dictionary (or map) of key-value field pairs. The keys are strings that identify the field. The value of a field can be one of the following *native* types;

- TextValue (String)
- LongValue (8-byte integral)
- DoubleValue (8-byte floating point)
- BlobValue (byte[])
- sub-map

This allows a full range of data attributes to be modelled using a record.

Records are identified by a name. The name of a record is unique within its declaring *service* (see Platform Concepts for more on services).

Nesting

A record allows nesting of record structures by using a *sub-map* to represent the nested record structure. However, unlike a record, a sub-map does not allow further nesting. This means that nesting is limited to a depth of 1; a parent record with many child sub-maps.

A nesting depth of 1 using sub-maps provides a simple mechanism for describing 2 dimensional data structures. This is sufficient, for example, to represent the data in a database table. Furthermore it serves as a reminder that data models using simple structures is best practice.

Atomic Changes

An *atomic change* to a record defines a group of fields and their values that change in a single transactional action. Observers of records only receive atomic changes. The scope of an atomic change is defined by the publisher.

Atomic changes provide consistency for record changes, a natural delta (incremental change) mechanism for sending record changes and a means to control data publishing rates.

Example

To explain the concept further, consider the following "FooBar" record with this initial field state:

Record: FooBar	
Field1	0
Field2	0
Field3	0

Figure 1 Initial State

The following sequence of independent changes occurs to the record in the publisher;

1. Field1 value set to 1
2. Field2 removed
3. Field1 value set to 2
4. Field4 added with value 15

When the publisher publishes the atomic change to this record, all the independent changes above are combined into a single atomic change and distributed to observers.

Observers of the FooBar record see the record state change, atomically, from the initial state above to this below:

Record: FooBar	
Field1	2
Field3	0
Field4	15

Figure 2 Next State

The structure of the record, as seen by observers, changes in a single *atomic* action. At the observer end, the atomic change is applied to the local copy of the record resulting in the resolved, new state.

A publisher can, of course, publish an atomic change for every single field change to a record. However, this approach can be inefficient for rapidly changing data sets and causes unnecessary network traffic.

Image-on-join

The concept of 'image-on-join', in the context of real-time data systems, describes the pattern where a subscriber for data receives a current image of the data when it subscribes (or 'joins') and after that receives only the deltas (incremental changes) of the initial image. The key principle of image-on-join semantics is that the subscriber must start consuming deltas from the point-in-time that the image represents, otherwise the state of the data item is incoherent.

In the ClearConnect platform, all data model subscription follows the image-on-join pattern. The atomic changes to the records are the deltas. Records include a publish sequence number that is used to ensure received atomic changes are in the expected sequence. An out of sequence atomic change results in a re-sync operation for the record.

Platform Concepts

A ClearConnect platform is composed of the following types of processes:

Process type	Purpose
Platform Registry	The registry is the core of the platform. The registry has a name which identifies the platform instance. It knows about all platform services and how to connect to them. If the registry is lost, **existing data connections are unaffected**. When a registry restarts, all agents re-register all platform services with the registry.
Agent	An agent connects to one, and only one, registry and is used to create *Service* and *Proxy* components. Through the registry connection, the agent has visibility of all services on the platform. The agent registers the services it creates with the registry. In general there is one agent per Java runtime.
Service	A service creates records and publishes updates to the records. Atomic changes to the records are published to subscribers (service proxy instances). A service can also publish remote procedure calls (RPC) that connected service proxy instances can invoke.
Proxy	A service proxy connects to one, and only one, service and subscribes for real-time record updates from that service. The proxy only receives the changes to records it subscribes for. The proxy can also invoke RPCs on the connected service.

Table 1 Platform components

Note: a platform registry defines a single, distinct ClearConnect platform instance.

Note: a platform is defined by the agents, services and proxies that are connected to the platform registry.

The diagram below shows an example platform and the connection topology between these 4 process types. P**lease note:** a service can support **multiple** proxies. For compactness, this diagram is only showing single proxy connections.

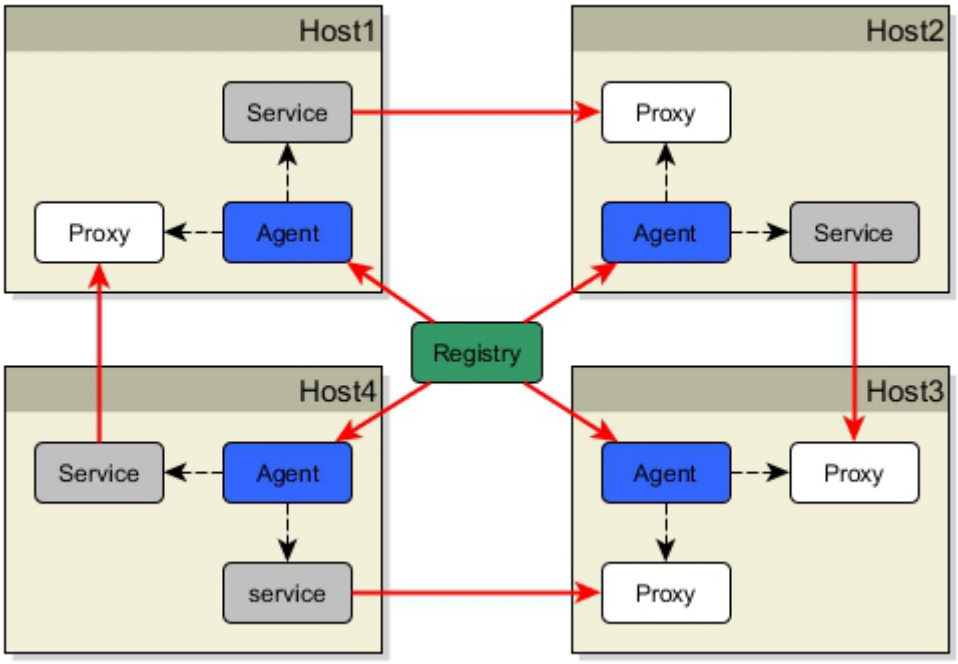


Figure 3 Platform Topology

As an observation, the Registry-Agent relationship follows the same behaviour as the Service-Proxy relationship; the agent subscribes for records from the registry and can (and does) invoke RPCs on the registry.

Data flow

Data flow is **one-way** between a service and proxy; from service to proxy. The service holds the records that change and distributes the changes to the connected proxies that have subscribed for the records. This reflects a general pattern that exists in all real-time data systems; data updates flow from publisher to subscriber.

In contrast to this, RPC invocation is in the opposite direction; from proxy to service. This also reflects another general pattern in real-time data systems where the publisher exposes control procedures that can be invoked by the subscribers.

Service Concepts

A service provides a logical grouping for records that all exist in the same domain. Application code creates a service and the records in the service, populates the records with data and publishes the atomic changes of the records.

Family, members, instances

A service is identified by a *service family name* that is unique on the platform. A service is a logical construct that is realised by one or more *service instances*. A service instance shares the family name of the service it supports and identifies itself within the family with a *member name*.

The diagram below illustrates this concept. This shows a service, family name “Foo”, composed of two instances with member names “Bar1” and “Bar2”. Each instance is uniquely identifiable by its service family name and member name (much like individuals in a family). Thus the “Foo” service exists on the platform, composed of two service instances “FooBar1” and “FooBar2”.

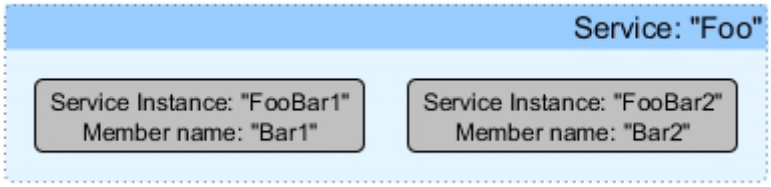


Figure 4 Service Family

Note: the term *instance* and *member* can be considered synonymous when discussed in the context of a service. Technically, only service instances exist on the platform and each service instance name must be unique on the platform. This is guaranteed as;

1. a service family name is unique on the platform
2. a member name is unique within a service family

A service (specifically an instance) contains records, all identified by a unique name within the service. Records can only be updated through the service instance. In no way can any record subscriber **directly** update a record. However, a service can publish an RPC that allows proxies to update records.

High-availability

A platform service provides fault-tolerance or load-balancing high-availability features. Both features are made possible due to a platform service being composed of one or more members.

The two features are mutually exclusive; a service either exists as a fault-tolerant service or a load-balanced service but not both. Additionally, all members of the service must be created with the same mode.

Mode	Description
Fault-tolerance	In this mode, one member in the service family is active, all other members are standby. The registry defines the active member. Should the active member drop off the platform, all proxy connections to the active member are routed to the next member in the family (as decided by the registry). This is a **warm-standby** fault-tolerance mechanism. This provides proxies with a seamless service connection even though the instance supporting the service may switch over. To achieve hot-standby, it is the responsibility of the application code to ensure that relevant record data is kept in-sync across all members of a fault-tolerant service. All service instances receive signals from the registry when they switch in/out of active/standby mode.
Load-balanced	This mode provides **connection** load-balancing across all members of a service. As proxy instances connect to the service, the registry assigns the next member in the service to be used, in a round-robin fashion. Should any of the service members drop off the platform, the connections it was managing are routed to the next available member, again in round-robin style. A service in this mode can expect to have duplicate subscriptions for records across its members.

Table 2 High-availability modes

Task Queue Concepts

The ClearConnect platform uses a bespoke task queue that makes efficient use of available threads. Real-time data systems require tasks in different contexts to be executed in-order; many systems designs end up having single-threaded executors to handle task processing for a single context, leading to a proliferation of threads that, most of the time, are idle.

The ClearConnect platform addresses this by having a pool of threads that execute tasks from a specialised task queue that manages two genres of tasks:

1. Sequential tasks; these are tasks that are maintained in sequence with respect to their *context*.
2. Coalescing tasks; these are tasks that can replace earlier, un-executed tasks, in the same *context*.

Both task genres declare the context they are bound to. These two task genres ensure in-order execution of tasks and skipping of old, out-of-date tasks using a single pool of threads. By sizing the thread pool to be the same number as the available cores a very efficient task processing engine exists that does not suffer from thread context switching.

The diagram below illustrates the composition of the task queue. The queue can be thought of as being a linked-list of *task contexts*. Each task context has its own internal queue of tasks that are bound to the same context. In the diagram, there are four task contexts in the queue diagram, three are sequential and one is coalescing (the diamond).

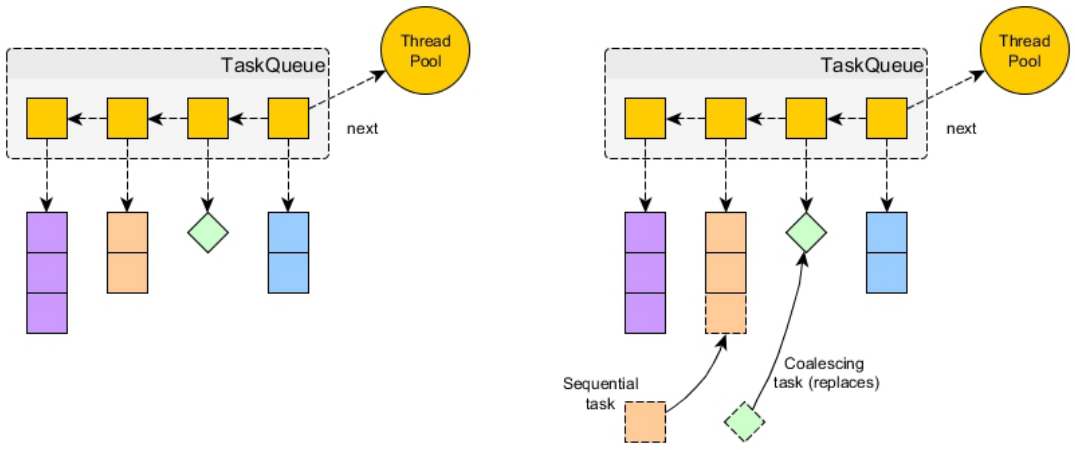


Figure 5 Task queue and adding tasks to the queue

When a sequential task is added to the task queue, it is added to the end of the internal queue for its matching task context. This ensures in-order execution of all the tasks in that context. In contrast, a coalescing task will simply overwrite its entry in its task context.

On execution of a sequential task context, the first task in the task context's internal queue is removed and executed by a thread in the pool then the entire task context is re-added to the back of the task queue and the next task context is taken from the queue.

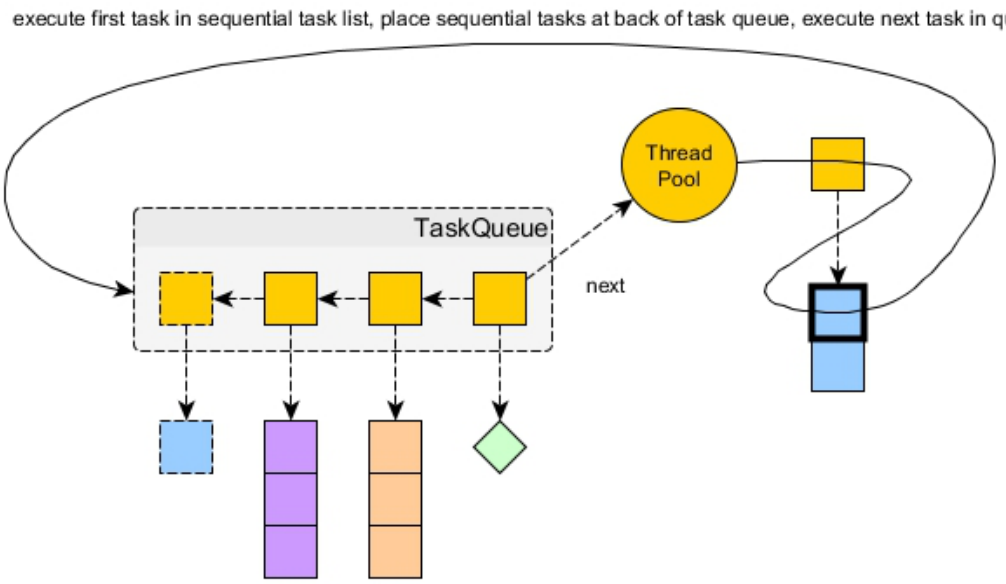


Figure 6 Task context executions

As the thread pool number increases, so does the through-put of task context execution and a maximum execution parallelism of task contexts equal to the number of threads in the pool.

Core Threads

The threads in the thread pool associated with the task queue are known as the "core threads" and are called *fission-coreX*, where X is the thread number. There is only one thread pool and task queue per *virtual machine* (VM). This is a key concept as it means that all proxy and service instances running in the same VM share the processing throughput of the core threads.

Event Stall

Blocking a core thread can cause an 'event stall' scenario where there are no more core threads available to process any events. The only methods where application code can possibly block a core thread are the `IRecordListener.onChange` and `IRpcExecutionHandler.execute` methods. So long as these are efficiently coded, an event stall cannot happen.

Programming Concepts

Technology Stack

The ClearConnect platform is designed to have zero dependencies on any 3rd party products. The technology stack is shown in the diagram below.

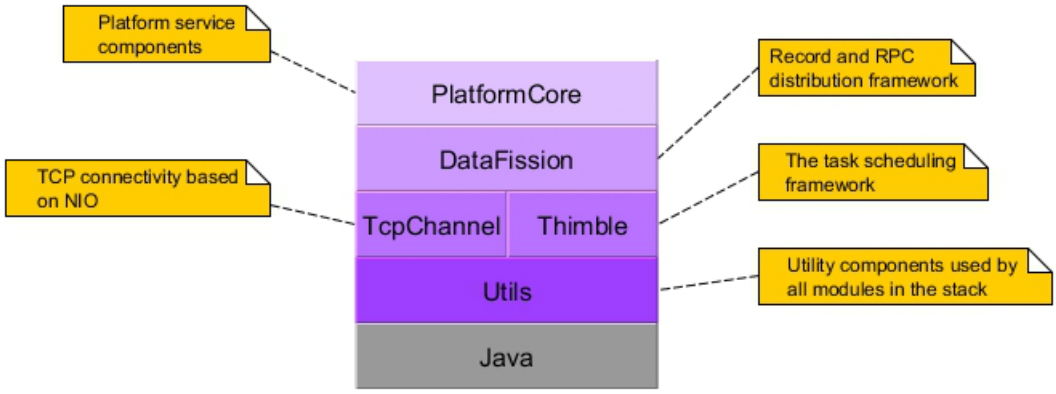


Figure 7: Technology Stack

EndPointAddress

ClearConnect is a distributed runtime environment with connections being made between proxies and services. Although the native network technology used by the platform is TCP, other technologies can be used (for instance Solace). An *EndPointAddress* is used to abstract the connection point of a service from the technology implementation specifics. In TCP terms, the end point address for a service is a server socket.

An end point address is composed of a *node* and a *port*. The node is typically the IP address of the host the service runs on. The port is a TCP port. However these two components of an endpoint can quite easily be translated into a topic-space address for a message-oriented technology.

Constructing Platform Components

The Platform Registry

A platform registry must be running for a ClearConnect platform to exist. To run a platform registry, construct a PlatformRegistry as in the example below:

```
EndPointAddress registryAddress = new EndPointAddress("127.0.0.1", 22222);

PlatformRegistry registry = new PlatformRegistry("Hello World Platform", registryAddress);
```

In general, the registry should run in its own virtual machine. As long as the registry process is running, the platform is running.

An Agent

An agent must be constructed in order to use the platform. To construct an agent you need two pieces of information:

1. The name of the agent
2. The *EndPointAddress* of the registry

The example below constructs an agent connecting to the registry specified previously:

```
EndPointAddress registryAddress = new EndPointAddress("127.0.0.1", 22222);

PlatformRegistryAgent agent = new PlatformRegistryAgent("Agent1", registryAddress);
```

In general, there should only be one agent constructed per virtual machine. The agent can be thought of as a session on the platform. The agent is used to construct (and destroy) services and proxies.

Platform Redundancy

An agent can provide multiple registry end point addresses on construction. This will force the agent to try each address, in order, until a successful connection is made. Should the agent lose the connection to the registry, the agent tries the next address in the list until a connection is made, looping through the list indefinitely.

When an agent re-connects to a registry, it re-registers all the services that were previously created though it (see next section). This mechanism allows a stand-by registry process to run and become aware of all platform services if the primary registry process is lost.

A Service

A service is constructed via an agent (technically, it is a platform service *instance* that is created). To construct a service the minimum following attributes are needed:

1. Service family name
2. Service member name
3. End point address node name (IP address of the host)
4. The wire protocol (the format used for sending messages)
5. The redundancy mode (HA feature)

The example below constructs a service using an agent:

```
// create the service
agent.createPlatformServiceInstance("Foo", "Bar1", "127.0.0.1", WireProtocolEnum.STRING,
RedundancyModeEnum.FAULT_TOLERANT);

// obtain the service
IPlatformServiceInstance serviceInstance = agent.getPlatformServiceInstance("Foo",
"Bar1");
```

Note that the agent maintains a reference to any created platform service instance. The reference is maintained until the service instance is destroyed via `agent.destroyPlatformServiceInstance(…)`.

A Proxy

A proxy to a service is also constructed via an agent. To construct the proxy, only the platform service family name is required. The agent performs all the work of contacting the registry to find out which instance to connect to.

The example below constructs a proxy to a service:

```
// wait to ensure the service is available (this will wait for 60 seconds at most)
agent.waitForPlatformService("Foo");

// obtain a proxy to the service
IPlatformServiceProxy proxy = agent.getPlatformServiceProxy("Foo");
```

As with a service, the proxy reference is maintained by the agent until it is destroyed using `agent.destroyPlatformServiceProxy(…)`.

Creating and publishing a record

A record is created and published using a service. In the example below, a record is created in the service, updated and published:

```
// create the record
IRecord record = serviceInstance.getOrCreateRecord("FooBar");

// update the record
record.put("message", "Hello World");
record.put("time", new Date().toString());
record.put("pi", DoubleValue.valueOf(3.1415926535898));

// publish the record change
serviceInstance.publishRecord(record);
```

Listening for record changes

Changes to a record are observed by registering a listener against the name of the record in the proxy. The record listener contains the application logic to respond to record changes.

The example below registers a listener against the record created in the service example above:

```
IRecordListener listener = new IRecordListener()
{
  @Override
  public void onChange(IRecord imageValidInCallingThreadOnly, IRecordChange atomicChange)
  {
    // NOTE: this is the correct way to snapshot a record image in a listener
    ImmutableRecord mySnapshot = ImmutableRecord.snapshot(imageValidInCallingThreadOnly);
    System.out.println(mySnapshot);
  }
};

proxy.addRecordListener(listener, "FooBar");
```

(!) **The image of a record is immutable only within the context of the listener's `onChange` method. Outside of this method, the record image is subject to change.**

You should not cache the IRecord of the `onChange` method if you want to keep a snapshot of the image. The correct way to capture a snapshot of the record's image during the `onChange` method is by using the snapshot code below:

```
ImmutableRecord mySnapshot = ImmutableRecord.snapshot(imageValidInCallingThreadOnly);
```

Putting it all together: The Hello World Example

All the previous examples can be used to create a single working "Hello World" example:

```
EndPointAddress registryAddress = new EndPointAddress("127.0.0.1", 22222);

PlatformRegistry registry = new PlatformRegistry("Hello World Platform", registryAddress);

PlatformRegistryAgent agent = new PlatformRegistryAgent("Agent1", registryAddress);

// create the service
agent.createPlatformServiceInstance("Foo", "Bar1", "127.0.0.1", WireProtocolEnum.STRING,
RedundancyModeEnum.FAULT_TOLERANT);

// obtain the service
IPlatformServiceInstance serviceInstance = agent.getPlatformServiceInstance("Foo",
"Bar1");

// wait to ensure the service is available (this will wait for 60 seconds at most)
agent.waitForPlatformService("Foo");

// obtain a proxy to the service
IPlatformServiceProxy proxy = agent.getPlatformServiceProxy("Foo");

// create the record
IRecord record = serviceInstance.getOrCreateRecord("FooBar");

// update the record
record.put("message", "Hello World");
record.put("time", new Date().toString());
record.put("pi", DoubleValue.valueOf(3.1415926535898));

// publish the record change
serviceInstance.publishRecord(record);
IRecordListener listener = new IRecordListener()
{
  @Override
  public void onChange(IRecord imageValidInCallingThreadOnly, IRecordChange atomicChange)
  {
    // NOTE: this is the correct way to snapshot a record image in a listener
    ImmutableRecord mySnapshot = ImmutableRecord.snapshot(imageValidInCallingThreadOnly);
    System.out.println(mySnapshot);
  }
};

proxy.addRecordListener(listener, "FooBar");
```

Example output and discussion

The output from running this will be similar to this below:

```
(Immutable)Foo->Agent1-20141023-20:52:22:744@192.168.56.1|FooBar|1|{message=SHello
World,pi=D3.1415926535898,time=SThu Oct 23 20:52:23 BST 2014}|subMaps{}
```

This is the `toString` format of an `IRecord`. The format is:

```
[Record context]|[Record name]|[Record sequence]|[data]|[sub-map data]
```

In the above example we have:

Record context	(Immutable)Foo->Agent1-20141023-20:52:22:744@192.168.56.1
Record name	FooBar
Record sequence	1
Data	{message=SHello World,pi=D3.1415926535898,time=SThu Oct 23 20:52:23 BST 2014}
Sub-map data	subMaps{}

The record context describes the name of the service or proxy that the record was from. For a proxy, the format is:

```
[Service family name]->[Agent name]
```

The agent name in the example is `Agent1-20141023-20:52:22:744@192.168.56.1` which itself has a format:

```
[Agent name]-[yyyyMMdd-HH:mm:ss:SSS]@[local host IP]
```

This provides an automatic unique naming mechanism for any agent.

The data is displayed in its key-value `toString` form. The `toString` of an IValue has the format:

```
[Type-code][String value]
```

The type codes per record IValue class are:

Type code	IValue class
S	TextValue
D	DoubleValue
L	LongValue
B	BlobValue

Table 3 Type codes per IValue class

So the string `message=SHello World` is self-describing; the key is "message", the data is a `TextValue` with string value of "Hello World". Similarly, the string `pi=D3.1415926535898` describes the key "pi" has data that is a DoubleValue with string value "3.1415926535898".

Note; any `IValue` class can be constructed from its string representation as in the example below:

```
DoubleValue pi = AbstractValue.constructFromStringValue("D3.1415926535898");
```

This concludes the discussion of the output for the "Hello World" example. More advanced programming concepts follow this.

Advanced Programming

This section discusses more advanced API concepts, classes and features available in the ClearConnect platform. The Javadoc of the API classes also provides further descriptions.

Ephemeral ports

When a service instance is constructed with no TCP port specified (or 0) then an *ephemeral* port is used. This allows the network layer to allocate available ports for the service without application configuration (a big plus).

Wire Protocol

ClearConnect has various built-in wire protocols that encode and decode data for different use-cases. A service is declared to use a wire protocol. Services are free to choose what protocol they want so the ClearConnect platform can have services using different wire protocols.

Wire Protocols	
STRING	A UTF-8 based wire format. This is the default.
GZIP	An ISO-8859-1 variation of the STRING wire protocol and uses a gzip algorithm to compress the wire format.
SYMBOL	An ISO-8859-1 based wire format that uses a string symbol table for record name and key names, only sending the symbol not the name. Additionally, doubles and longs are sent in a binary format to minimise transmission. This is the most efficient transfer protocol.
CIPHER	An encrypted STRING wire protocol using AES 128bit encryption.

Event Listeners

Distributed systems are subject to disruption, the data sets and functions they can provide are also volatile. Without relevant signals to detect these situations, a component in a distributed environment is effectively blind to what is available. The ClearConnect platform provides an extensive set of event listeners to detect the various platform signals that represent service, data and function availability.
NOTE: All event listeners are executed by the *core* threads.

Listener	Registration Scope	Purpose
`IRecordAvailableListener`	Proxy/Service	Notifies when records are created in a service (and thus being available)
`IRecordSubscriptionListener`	Proxy/Service	Notifies when a record is subscribed for. Knowing whether any process has subscribed for a record means the data does not need to be sourced until the first subscriber appears.
`IRpcAvailableListener`	Proxy/Service	Notifies when an RPC is created in a service (and thus available to be called by a proxy)

IFtStatusListener	Service	Invoked when a service instance becomes active or standby if it is part of a fault tolerant service family.
IProxyConnectionListener	Service	Provides a service with the ability to be notified when new proxy connections are made to it.
IRecordConnectionStatusListener	Proxy	Notifies when a record is connected, disconnected or reconnecting. This is a mechanism for identifying if data in a record should be considered *live* or *stale*.
IServiceConnectionStatusListener	Proxy	Notifies when the connection to a proxy's service is connected, disconnected or reconnecting.
IRegistryAvailableListener	Agent	Notifies when the registry is available or unavailable.
IServiceAvailableListener	Agent	Notifies when a service family becomes available or unavailable.
IServiceInstanceAvailableListener	Agent	Notifies when a service instance becomes available or unavailable.

Table 4 Event listeners

RPC

Services expose behaviour by publishing RPC instances that proxies can invoke. The example below shows a service registering an RPC and a proxy invoking it:

```
// Construct the registry, agent, service and proxy
EndPointAddress registryAddress = new EndPointAddress("127.0.0.1", 22222);
PlatformRegistry registry = new PlatformRegistry("Hello World Platform", registryAddress);
PlatformRegistryAgent agent = new PlatformRegistryAgent("Agent1", registryAddress);
agent.createPlatformServiceInstance("Foo", "Bar1", "127.0.0.1", 22223,
WireProtocolEnum.STRING, RedundancyModeEnum.FAULT_TOLERANT);
IPlatformServiceInstance serviceInstance = agent.getPlatformServiceInstance("Foo",
"Bar1");
agent.waitForPlatformService("Foo");
IPlatformServiceProxy proxy = agent.getPlatformServiceProxy("Foo");

// Construct the RPC
IRpcInstance rpc = new RpcInstance(new IRpcExecutionHandler()
{
  @Override
  public IValue execute(IValue... args) throws TimeOutException, ExecutionException
  {
    // this is the 'body' of the RPC,
    // in this case just returning a string that echos
    // the first received argument + "world!"
    return TextValue.valueOf(args[0].textValue() + "world!");
  }
}, TypeEnum.TEXT, "helloWorldRpc", TypeEnum.TEXT);

// publish the RPC
serviceInstance.publishRPC(rpc);
```

```
// Call the RPC
IValue result = proxy.executeRpc(1000, "helloWorldRpc", TextValue.valueOf("hello"));

System.out.println("RPC result: " + result.textValue());
```

The result from executing this is:

```
RPC result: helloworld!
```

2-phase timeout

One of the features of the RPC mechanism in the ClearConnect platform is its 2-phase timeout. The phases for a timeout are:

Phase	Description
Execution start	This is the time between invoking in the proxy and the RPC execution starting in the remote service.
Execution duration	This is the time allowed for the execution to complete after it has started in the remote service.

Table 5 RPC execution phases

The timeouts are defined when executing the RPC, by default they are both 5 seconds.

Threading issues for RPCs

WARNING: if an RPC is invoked from the body of the `IRecordListener.onChange` method or the body of the `IRpcExecutionHandler.execute` method, there is an *event stall* danger.
However, it is **always** safe to execute an RPC using the `executeNoReponse` method as there is no response and thus no thread blocking.

Handling fast-producer scenarios

A proxy has no control over the record publishing rate of the service it is connected to. To solve the situation where the data rates are too fast for application code in the `IRecordListener.onChange` to process efficiently, the listener can be replaced with a `CoalescingRecordListener`. This provides a means to consume the updates from the service, batch them up and essentially produce an atomic change representing multiple atomic changes made to the record.

However the `CoalescingRecordListener` cannot be used if the service publishes discrete changes that cannot be coalesced into a single, bigger change.

Access Control

The ClearConnect API includes features for controlling access to services and data within services.

Network Access Control

When using the TCP as the transport technology, a service can be configured with a *whitelist* of acceptable inbound IP addresses. This is configured as a system property and specifies a comma-separated list of regular expressions to match against IP addresses. Note

that because they are regular expressions a literal "." must be escaped with "\". Any socket connections that do not match the whitelist will be terminated before any messages are processed.

```
-DtcpChannel.serverAcl=10\.0\.0\.*;10\.1\.2\.3
```

Session Access Control

All connections between a proxy and service include a session that is "synchronised" as the first order of business. This allows session-level access control. As part of the synchronisation, a proxy sends over a set of string attributes that are inspected by the receiving service. If the attributes are valid, the service responds with a session ID, otherwise the link is terminated.

The following classes and interfaces are used for session access control:

Class/Interface	Description
SessionContexts	Central component used to register
ISessionManager	Receives attributes from a proxy and validates them to create a session ID.
ISessionAttributesProvider	Provides the attributes to send to a service (e.g. username and password).

Note: only when using the CIPHER wire protocol will all session attributes and responses be encrypted over the wire. All other wire protocols have no encryption of session information.

Data Access Control

When subscribing for a record, a proxy can provide a permission token. The permission token is a string that has meaning in the application domain. The subscription is received by the service along with the permission token and application code can be invoked to inspect the token and assess whether the subscription should be allowed by the proxy. Typically, this mechanism is used to ensure clients subscribe for data they are entitled for.

This code snippet helps to demonstrate this

```
service.setPermissionFilter(new IPermissionFilter()
{
      @Override
      public boolean accept(String permissionToken, String recordName)
      {
            // examine the token and record...
            // return false if the subscription is not allowed
            // return true if the subscription is OK
            return false;
      }
});
```

Utility Classes

The ClearConnect platform does not depend on any 3rd party library. This removes any compatibility problems outside of the JDK and it has also allowed us to ensure that specific and optimised code is used throughout. To achieve this, the platform has a suite of utility classes that all reside in the `com.fimtra.util` package. These contain public static methods that are are not exposed explicitly in the API but are useful for any application that uses the platform.

All the public static methods have complete javadoc that should be used for reference. Also, some of the classes are discussed in the following sections.

The Log Utility Class

The `Log` class is used for all logging that occurs at runtime in the platform (see the Log files section on page 25). The log methods result in statements being written to the messages log file asynchronously.

Unlike many logging frameworks there are no logging levels. Everything is logged because the messages log file contains the minimum yet complete information to diagnose any issues that occur in the platform. Omitting logging levels makes logging less complex and faster.

If you choose to use the platform Log utility, it is best practice to use the `log(Object source, String... messages)` where possible because it is the most efficient one. Also it is recommended that you follow the same principle uses in the platform itself; log concisely and only what is essential.

The ThreadUtils Utility Class

The platform has a powerful threading model that reduces the need for a developer to use additional threads, however there may be situations where additional threads are needed.

The `ThreadUtils` class has a number of `Thread` and `Executor` creation methods. Using them means the strict thread naming approach used in the platform will be adopted. This gives you the opportunity to use the names in log statements to facilitate diagnosis of issues. If you also choose to use the platform's log utility, thread names are automatically logged with every statement.

Runtime Concepts

Log files

At runtime, the VM running any ClearConnect platform component produces 3 log files that reside in the *logs* directory (configurable) of the working directory of the VM. The logs directory is automatically created. All the files automatically roll-over after 1M (configurable) and automatically delete files older than 24hrs on VM startup.

Log file name	Purpose
`[main-classname]-`**`messages`**`_yyyyMMdd_HHmmss.log`	Platform messages and events are written to this file.
`[main-classname]-`**`threaddump`**`_yyyyMMdd_HHmmss.log`	Periodic thread dumps are written to this file.
`[main-classname]-`**`Qstats`**`_yyyyMMdd_HHmmss.log`	Periodic core task queue statistics are written to this file.

Table 6: Log file types

e.g. `RemoteTestRunner-messages_20141020_200504.log` is the log file created when running the JUnit tests (the main class is RemoteTestRunner)

Logging Format

The platform messages log file has a format intended to provide in depth and useful runtime information when the logging occurred. The aspects of the logging are outlined by dissecting the log statement below:

```
20141028-09:21:27:882    fission-core3-0    com.fimtra.datafission.core.Context:1255364991
Notifying initial image 'Services'
```

Log message component	Description
`20141028-09:21:27:882`	The timestamp when the log message occurred, format: `yyyMMdd-HH:mm:ss.SSS`
`fission-core3-0`	The thread name
`com.fimtra.datafission.core.Context:1255364991`	The fully qualified class followed by the hashcode of the class instance
`Notifying initial image 'Services'`	The log statement

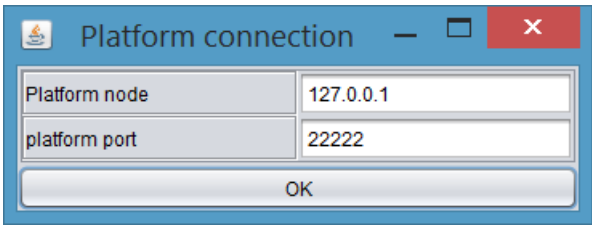
Table 7: Log message components

Monitoring

Platform Desktop

The ClearConnect platform has a built in graphical interface tool for investigating and visualising all services and data running on the platform. The tool is called the 'platform desktop' and is run by executing the main method of the class `com.fimtra.clearconnect.core.PlatformDesktop`

When it starts, the user is prompted to enter endpoint information for the platform that it will connect to:



The screenshot below shows the desktop running on a platform called “TestPlatform”. The interface uses a context menu (right-click menu) user interaction model. Right-clicking on the desktop shows the first level navigation options.

Figure 8: PlatformDesktop

Selecting any of these options opens a window showing the view selected. The rows in each view may have further context menu navigation to open up further windows showing context information of the selected row. This allows a drill-down investigation of services and data in the platform. A full map of navigation of the various views is shown later on.

Behaviour

- All data in the desktop is live data on the platform.
- The view windows can be resized, repositioned within the confines of the desktop.
- New windows always open at the top right and may obscure existing windows until they are repositioned.
- Columns in each view can be re-ordered and re-sized.
- Clicking on the column header will sort the rows in ascending/descending order for the data in the column. Sorting does not occur on updates of data (the rows will not auto order as data changes).
- Changes in any cell value are highlighted by the background flashing red for 1 second then reverting.

- The desktop frame, size, position and the internal view positions, sizes, column orders, column widths are all saved on exit. The layout information is held in a file in the working-directory called `PlatformMetaDataModel_[platform name].ini`. This allows multiple platforms to have different layouts saved.
- When viewing records, if the service disconnects the connection will automatically be retried until the service returns or the window closed. The table background colour signals the state:
  - Yellow – retrying connection
  - Orange – disconnected

Note: sub-maps viewable by double-clicking on the "SubMap..." text.

Platform Desktop Navigation

The diagram below illustrates the navigation drill-down paths available in the `PlatformDesktop`. The views are described in the next section.

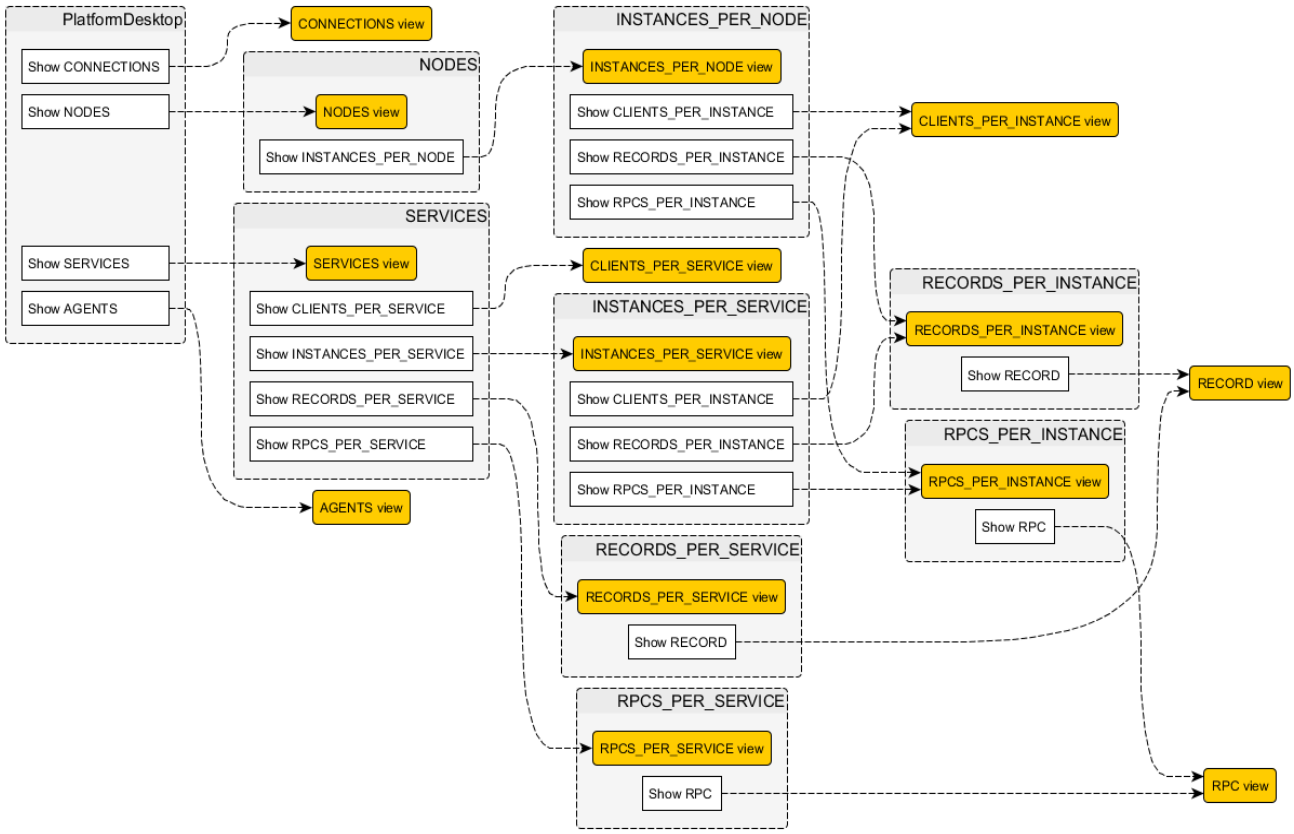


Figure 9: PlatformDesktop navigation

Platform Desktop Views

The table below describes each of the views and the attributes associated with each one. Generally, statistic-type attributes update at 10 second intervals (e.g. messages received, Kb received). Services (and instances) appear as soon as they are registered with the platform registry. Services disappear from desktop views when the platform registry detects that the service instance is no longer available (so this can be subject to heartbeats being missed before the service is removed, these are discussed later on).

View	Purpose
CONNECTIONS	This view shows all connections on the platform. A connection is from a publisher (service) to a client (proxy). The connection naming syntax is *[service]->[proxy]* Agent connections are called PlatformRegistry->*[agent name]* Attributes: • Name of the connection. • Publisher node and port. • Wire protocol for the connection. • Subscriptions the client has with the service. • Number of messages published to the client. • Kb count received by the client. • Connection uptime (seconds).
DES	This view shows all nodes that have a service instance running. It provides a physical machine view of the platform. Attributes: • Name of the node (IP address). • Number of service instances running on the node.
SERVICES	Shows all the service families running on the platform. Attributes: • Name of the service family. • The redundancy mode of the service family. • The number of instances (members) running in this family. • The number of connections the service family is supporting (summed over all instances). • The number of RPCs and records the service family has. This number is the sum of all distinct named records/RPCs in all service family instances.
AGENTS	Shows all agents running on the platform. In general, there should be one agent per VM (although this is not enforced). The agents view provides an insight into the runtime that hosts one (or more) services. Attributes: • Name of the agent. • Agent uptime. • Number of CPUs for the host. • Number of threads in the JVM. • Memory in use/available in the JVM. • OS and JVM versions. • Process name. • Task queue total submitted. • Task queue overflow. The overflow should be 0 under calm conditions. If this stays greater than 0 for a sustained period, this indicates that there are too many events occurring and not being processed fast enough. • Events per minute (EPM). This shows the number of events being processed in the task queue per minute (extrapolated from a 30 sec sample). Numbers in the 100,000 are busy task queues. This queue number

	represents the event activity of all services in the agent's runtime.
`INSTANCES_PER_NODE`	Shows the service instances running on a single node. Attributes • Name of the service instance. • Service instance node and port. • Uptime of the service. • Number of RPCs and records in the service. • Number of connections to the service. • Wire protocol used by the service. • Number of client connections. • Number of messages published to all clients. • Kb count published to all clients. • The name of the agent that created it.
`CLIENTS_PER_SERVICE`	Shows the clients connected to a service family. This includes all service instances for the family so provides a grouped connection view per service family. Attributes: • Client connection name: `[instance]->[proxy]`. • Client endpoint. • Service instance endpoint. • Subscriptions the client has with the service. • Number of messages published to the client. • Kb count received by the client. • Connection uptime (seconds).
`INSTANCES_PER_SERVICE`	Shows the all the running members (instances) for a service family. Similar to the `INSTANCES_PER_NODE` view. Attributes: • Name of the service instance. • Service instance node and port. • Uptime of the service. • Number of RPCs and records in the service. • Number of connections to the service. • Wire protocol used by the service. • Number of client connections. • Number of messages published to all clients. • Kb count published to all clients. • The name of the agent that created it.
`RECORDS_PER_SERVICE`	Shows all the records in a service family. This is an aggregated view the records across all running member instances of the service family. Attributes: • Name of each record. • Number of subscriptions for each record.
`RPCS_PER_SERVICE`	Shows all the RPCs in a service family. This is an aggregated view of RPCs across all running member instances of the service family. Attributes: • Name of the RPC. • RPC definition (arguments and return value type).

CLIENTS_PER_INSTANCE	Shows the clients attached to a service instance. One row per client connection. Attributes: • Name of the connection [service]->[client]. • Service endpoint, client endpoint. • Connection uptime. • Service family name. • Service instance name. • Messages received. • Kb count received. • Subscriptions with service.
RECORDS_PER_INSTANCE	Shows all the records in a service instance. Attributes: • Name of each record. • Number of subscriptions for each record.
RPCS_PER_INSTANCE	Shows all the RPCs in a service instance. Attributes: • Name of the RPC. • RPC definition (arguments and return value type).
RECORD	Shows each field of a record as a row in the view. Each column is a different record instance.
RPC	Allows an RPC to be invoked. Text values per RPC argument.

Table 8: PlatformDesktop View descriptions

Deadlock Detection

Every agent runtime includes a deadlock detection feature. Any deadlocks discovered (even in application code running in the same VM) are logged to the `messages` log file.

Connection Monitoring

All connections between a service and proxy are monitored for aliveness. This is achieved by every VM having a `ConnectionWatchdog` that monitors connections. When a connection is established, the `ConnectionWatchdog` starts to monitor it. The watchdog periodically sends a heartbeat message to all registered connections and checks each connection to verify that a heartbeat (or data) has been received from the other end of the connection (received data can count as a heartbeat). If no heartbeat or data has been received for a specified number of periods (default 3, period default 5000ms) then the watchdog closes the connection. If a heartbeat cannot be sent, the watchdog also closes the connection.

Service Instance Monitoring

Service instances on the platform are all individually monitored by the `PlatformRegistry`. This is achieved by the registry acting as a client to each service and subscribing for the system records that each service provides. This provides the registry with knowledge about what activity is occurring in each service instance (via the system records, described later on) and whether the service instance is available (via the connection being available). If the service instance is stopped, the registry will detect this when the connection closes or the watchdog detects the connection is not alive. Once the registry loses a service instance connection it does not attempt to re-establish the connection. The service must be re-registered with the registry via the agent.

How Connections are Initiated

Connections are initiated by the proxy asking the agent for connection details for the service family. The agent does this by requesting a 'service info record' from the registry for the service family. The service info record includes all the parameters needed to connect to a service instance for the requested service family. The proxy then makes a connection using this information. If there is no service info record, the proxy will pause for 5000ms before asking the agent again. This cycle continues until either:

a) the proxy connects
b) the proxy is destroyed code via a call to `agent.destroyPlatformServiceProxy(...)`.

Connection activity is always logged in the `messages` log (example below from a `PlatformDesktop` connecting to the registry). Connections **always** start off in a `DISCONNECTED` state.

```
20141029-11:33:56:813    main    com.fimtra.datafission.core.Context:369134045
DISCONNECTED PlatformRegistry->PlatformMetaDataModel-20141029-11:33:56:541
20141029-11:33:56:820    main    com.fimtra.datafission.core.ProxyContext:208924888
Constructing channel using TcpChannelBuilder [host=127.0.0.1, port=22222,
frameEncodingFormat=TERMINATOR_BASED]
20141029-11:33:56:841    main    com.fimtra.channel.ChannelWatchdog:2060219449
Heartbeat period is 5000ms, missed heartbeat count is 3
20141029-11:33:56:843    tcp-channel-reader    com.fimtra.tcpchannel.TcpChannel:1189961117
Connected java.nio.channels.SocketChannel[connected local=/127.0.0.1:61048
remote=/127.0.0.1:22222]
20141029-11:33:56:843    main    com.fimtra.tcpchannel.TcpChannel:1189961117
Constructed TcpChannel [connected /127.0.0.1:61048->127.0.0.1:22222]
```

```
20141029-11:33:56:846    fission-core3-0    com.fimtra.datafission.core.Context:369134045
CONNECTED PlatformRegistry->PlatformMetaDataModel-20141029-11:33:56:541
```

Connection Auto-Reconnect

If a connection between a service and a proxy closes unexpectedly (i.e. without the proxy being destroyed), the proxy re-initiates the connection to the service. When the connection is re-established, the proxy re-subscribes for all records that were previously subscribed for. This mechanism ensures that auto-reconnect occurs and also ensures that all data in the proxy is refreshed and is live.

Service System Records

Every service has five special records called 'system records'. These capture information about the state of the service. Typically application code does not need to access these (the Event Listeners provide the same level of information). System records are read-only for application code.

System record name	Purpose
`ContextRecords`	Holds the name of all the records created.
`ContextSubscriptions`	Holds the number of subscriptions per record.
`ContextRpcs`	Holds the RPCs published.
`ContextConnections`	Holds all connections and the connection attributes.
`ContextStatus`	Holds other status attribute information.

Table 9: System records

License

ClearConnect is covered by Apache License 2.0. This gives you the right to use ClearConnect for private or commercial purposes without paying any royalties.
You also have the right to modify the code and to warranty and license your product.
You cannot use the name ClearConnect for your derivatives or hold fimtra or its officers liable for any defect.
If you modify the code you must include the original copyright, a copy of the original license and Notice file and state any changes you have made to the software.

Enterprise Development

In order to facilitate development of enterprise applications a higher level API is available. This provides a more powerful way of using the ClearConnect platform. The API consists of three objects that provide access to the platform, they are listed below.

Access object	Purpose
`PlatformKernel`	Used to start a registry and a config service in a single VM.
`PlatformClientAccess`	Provides enterprise access to the platform for a client (subscriber).
`PlatformServiceAccess`	Provides enterprise access to the platform for a service (publisher).

Table 10: Enterprise access objects

Config Service

The `PlatformKernel` starts a registry and a config service in a single VM. The config service can be used for configuring any client or service that is on the platform. Although the config service is actually a platform service the `PlatformKernel`, `PlatformClientAccess` and `PlatformServiceAccess` provide simplified access to it in the form of a `IConfigServiceProxy`. This in turn has a mechanism for reading configuration and a mechanism for modifying configuration.

Reading Configuration

Configuration for a service family and member can be read on demand or subscribed for, both via the `IConfig` object. This allows for configuration to be applied as part of specific functionality (for example initialisation at start-up) or dynamically during the life cycle of an application.

When configuration is read, it is layered so that it is possible to override it in a flexible manner. This layering takes place as outlined below.

Config type	Provided by	Priority
Local	Filesystem: property files located in the *config* directory (configurable)	1 (overrides priorities 2 and 3)
Member	Config service	2 (overrides priority 3)
Family	Config service	3

Table 11: Configuration layering

Modifying Configuration

Configuration can be modified using the `IConfigManager` that is obtained using the client or service family and member names. Methods to create/update and delete configuration for the family and member configuration are available. These are actually RPC's that are handled by the config service. However it is not possible to manage local properties in this manner, local properties should be edited directly on the filesystem.

Persistence of Configuration

The config service handles the persistence of the configuration. Internally the configuration are records, when an RPC is called on the config service that modifies the configuration, it saves the record state to disk and then publishes the new state.

On startup the config service resolves persisted records form the record files in order to recover the configuration state.

Customising Persistence of Configuration

By default configuration is persisted to disk, however any custom persistence mechanism (for example to a database) can be plugged in. This is done by

1. implementing the `IConfigPersist` interface
2. specifying the fully qualified implementation class using the system property `platform.configService.configPersistClass`. Note that the implementation class has to have a public, no argument constructor.

Configuration Example

Here is a worked example to illustrate intended use of configuration. There is a service on the platform with family name *HelloWorld* and member name *Active*. It has been implemented to read configuration at start-up.

The following property for the *HelloWorld* service family is added:
service.updateFrequencySeconds=5
In code this is done using

```
IConfigManager.createOrUpdateFamilyConfig("service.updateFrequencySeconds", 5)
```

During startup the *Active* member gets its config, for key **service.updateFrequencySeconds** there is a value of 5.

Then the following property for the *Active* member is added:
service.updateFrequencySeconds=1
In code this is done using

```
IConfigManager.createOrUpdateMemberConfig("service.updateFrequencySeconds", 1)
```

On restart, the *Active* member gets its config for key **service.updateFrequencySeconds**, this time it has a value of 1.

Finally a file located in the *config* directory, localOverrides.properties has this line added:
service.updateFrequencySecords=2
The next time the *Active* member is started it will have a config value of 2 for the key **service.updateFrequencySecords**.

PlatformClientAccess And PlatformServiceAccess

The `PlatformClientAccess` and `PlatformServiceAccess` objects provide access to the platform for clients and services respectively. Both objects provide methods to obtain the platform agent and the config service proxy (`IConfigServiceProxy`), in addition `PlatformServiceAccess` provides access to the service proxy of the service itself.