Table of Contents

Configuration

One of the utilities the *MadgikCommons* library offers is the Configuration utility. Its main purpose is to serve configuration values to the various clients that are using it but as will become apparent in the following paragraphs, it can be used to offer additional functionality. One of the key features of the Configuration Utilities is that is provides *read only* access to the values it exposes. If a client needs to store information that will be available to later calls, the State Utility is the one to use.

Configuration file

The configuration file is the location where the parameters that are needed to be shared through the *ConfigurationManager* are declared. The location of the configuration file is currently the directory tht the application is running. This is something to be revised in following versions. A number of alternatives can be considered such as

- the location is passed using traditional .properties files
- a number of predefined locations are scanned with a specific order
- an environmental variable defines the location

One, all or a conjunction of the above alternatives can be considered to define the location of the configuration file.

The schema of the configuration file is the following:

<configuration>
<param ..... > value </param>
<param ..... > value </param>
<param ..... > value </param>
<param ..... > value </param>
</configuration>

The configuration file is simply a list of parameters. Each parameter defines a configuration value the *ConfigurationManager* will be serving. Each parameter depending on its type can have a different type of value. The parameter types that are available are the following:

- *IntegerClass*
- *IntegerPrimitive*
- *FloatClass*
- *FloatPrimitive*
- *DoubleClass*
- *DoublePrimitive*
- *ShortClass*
- *ShortPrimitive*
- *LongClass*
- *LongPrimitive*
- *BooleanClass*
- *BooleanPrimitive*
- *ByteClass*
- *BytePrimitive*

- *String*
- *XML*
- *Object*

For the available primitives two different types are defined. One representing the boxed, and the other the un-boxed type (Integer and int). This is needed because of the way the java framework treats method signatures. As will become apparent later on, a parameter of a specific type can be used to instantiate an object, call a specific class method etc. This is done through reflection which uses argument types to discover the method that needs to be called. In case a method declares its signature using a boxed type, it cannot be discovered using the un-boxed type and vise versa. This requires both types, boxed and un-boxed to be specifically declared in the configuration parameter declaration. Additionally, the *XML* type is basically the same with the *String* type with the difference that is performs some basic escaping and un-escaping of the XML invalid characters. The *Object* type is the most interesting of the available types as it represent a specific java class that can be instantiated, manipulated, have one or more of its methods called and set values to other parameters.

Each of the parameter elements contain a number of attributes that define the name, type and behavior of the parameter. These attributes are the following:

- name – Is the unique name with which any client can retrieve the value of the parameter.
- type – The type of the parameter. This must be one of the types listed above.
- generated – This value indicates whether the *param* element contains the value of the parameter or the value will be generated by another parameter during its evaluation.
- internal – This value indicates the visibility modifier of the parameter. An internal parameter cannot be seen by any of the clients.
- shared – This attribute is applicable only in parameters of type *Object*. The default behavior of all parameters is to be served in a a non shared fashion. This means that if a client retrieves a value, it will be its own local copy of the value. Changes it makes to this value will not be reflected in other clients copies. For the *Object* type there is also the option to share the instance that is served between the clients. Event though this possibility is given, the Configuration utility does not provide any locking or synchronization methods to access the object. This should be handled externally either by the clients or by the object itself. Additionally to having this attribute set, the object that

Any modifications to the values exposed by the Configuration Utility are transient. Once the *ConfigurationManager* is reinitialized, the changes will be lost. Initialization only takes pace once when the class is first loaded.

An example of a configuration file for all the above types of parameters is the following:

```
<parameters>
  <param name="parameterKey1" type="IntegerClass" generated="false" internal="false">4</param>
  <param name="parameterKey2" type="IntegerPrimitive" generated="false" internal="false">4</param>
  <param name="parameterKey3" type="FloatClass" generated="false" internal="false">4.5</param>
  <param name="parameterKey4" type="FloatPrimitive" generated="false" internal="false">4.5</param>
  <param name="parameterKey5" type="DoubleClass" generated="false" internal="false">4.5</param>
  <param name="parameterKey6" type="DoublePrimitive" generated="false" internal="false">4.5</param>
  <param name="parameterKey7" type="ShortClass" generated="false" internal="false">4</param>
  <param name="parameterKey8" type="ShortPrimitive" generated="false" internal="false">4</param>
  <param name="parameterKey9" type="LongClass" generated="false" internal="false">4</param>
```

```
  <param name="parameterKey10" type="LongPrimitive" generated="false" internal="false">4</param>
  <param name="parameterKey11" type="BooleanClass" generated="false" internal="false">true</param>
  <param name="parameterKey12" type="BooleanPrimitive" generated="false" internal="false">true</param>
  <param name="parameterKey13" type="ByteClass" generated="false" internal="false">0</param>
  <param name="parameterKey14" type="BytePrimitive" generated="false" internal="false">0</param>
  <param name="parameterKey15" type="String" generated="false" internal="false">Hello World</param>
  <param name="parameterKey16" type="XML" generated="false" internal="true">&lt;x/&gt;</param>
  <param name="parameterKey17 type="String" generated="true" internal="false"/>
  <param name="parameterKey18 type="Object" generated="true" internal="false" shared="true"/>
  <param name="parameterKey19" type="Object" generated="false" internal="false" shared="false">
   <class value="full.package.name.ClassName"/>
   <constructor>
    <arguments>
     <arg order="1" name="arg1" param="parameterKey1"/>
     <arg order="2" name="arg2" param="parameterKey2"/>
     <arg order="3" name="arg3" param="parameterKey3"/>
    </arguments>
   </constructor>
   <calls>
    <method order="1" name="method1">
     <arguments>
      <arg order="1" name="arg1" param="parameterKey4"/>
      <arg order="2" name="arg2" param="parameterKey5"/>
      <arg order="3" name="arg3" param="parameterKey6"/>
     </arguments>
     <output param="parameterKey17"/>
    </method>
    <method order="2" name="method2">
     <arguments>
      <arg order="1" name="arg1" param="parameterKey8"/>
      <arg order="2" name="arg2" param="parameterKey9"/>
      <arg order="3" name="arg3" param="parameterKey10"/>
     </arguments>
     <output param="parameterKey18"/>
    </method>
   </calls>
  </param>
</parameters>
```

Dependency resolution

As it is obvious from the above example, a parameter can depend on other parameters. In the above example in order to instantiate as describe the object parameter with name *parameterKey19*, the parameters with names *parameterKey1*, *parameterKey2*, *parameterKey3*, *parameterKey4*, *parameterKey5*, *parameterKey6*, *parameterKey8*, *parameterKey9*, *parameterKey10* must already be available. Additionally, the parameters with names *parameterKey17* and *parameterKey18* are marked as generated parameters, which means that the instantiation and evaluation of the parameter with name *parameterKey19* will assign them the values that will be retrieved from the invocation of the methods *method1* and *method2*. The dependency graph that is constructed is apparent.

On the initialization of the *ConfigurationManager*, once when the class is loaded, this dependency graph is constructed. Depending on its edges the evaluation of each parameter is performed.

Currently the nodes of the graph are the parameters so a dependency that is self referencing

cannot be resolved and will cause an initialization error. A self referencing dependency is for example when a method of an *Object* parameter needs as input a parameter that is generated by a method of the same *Object* parameter which is invoked with a smaller order. Even though this limitation is not considered very limiting since even in a case where this might appear there are workarounds such as an intermediate, possibly internal, auxiliary parameter, it might impose some design restrictions or even a costly workaround. In future versions the dependency resolving algorithm will be enhanced to contain also this granularity of dependency resolution.

State

Another utility the *MadgikCommons* library offers is the State Store Utility. This utility offers its clients the possibility to store key – value pairs which are persisted and can be reaccessed even after VM restarts and host reboots. The value part of the pair can be any of a number of different types and will be handled differently during storage and retrieval. The basic functionality this utility offers is at its bases common to the one the Configuration Utility offers. The key difference is that while the Configuration Utility is read only for its clients, the State Store Utility is both read and write.

This separation of usage types was made having in mind the scenarios this two kinds of usages are needed, the frequency that they would be normally appear in a client code and the scope of each. One would expect to have frequent usage of a Configuration value possibly with many appearances in the code. The purpose is to serve this with imposing minimum overhead as the Configuration Utility described above aims to. On the other hand, a State manager aims to serve needs that have to do with global application data. Data that would for example be created over a period of time and would be stored as part of a check-pointing mechanism so that if needed the application will not need to restart from the beginning.

Store Configuration

The State Store Utility uses the local filesystem to persist the state its clients register. To accomplish this the *StateManager* uses two files in the local filesystem whose usage it separates, and uses one as a registry file for storing metadata on the entries that are available, and another as a data file for storing the actual payload the clients provide. Additionally, for converting characters to bytes and vise versa for the serialization and deserialization operation, a character set needs also be configured. Another configuration parameter that is expected is whether or not the *StateManager* should cleanup any unneeded, invalid, entires from the registry as well as the data files.

The State Store utility relies on the Configuration Utility for these configuration values. More specifically an example of the configuration parameters that needs to be added in the configuration file is the following:

```
<param name="StateManager.CleanUpOnInit" type="BooleanPrimitive" generated="false" internal="false"> perform data and registry file clean up on initialization </param>
<param name="EncodingCharset" type="String" generated="false" internal="false"> charset name (eg.UTF-8)</param>
<param name="StateManager.EntryRegistryPathName" type="String" generated="false" internal="true">path to registry file (eg./tmp/registryfile.data)</param>
<param name="StateManager.EntryRegistryFile" type="Object" generated="false" internal="false" shared="false">
 <class value="java.io.File" />
 <constructor>
  <arguments>
   <arg order="1" name="pathname" param="StateManager.EntryRegistryPathName"/>
  </arguments>
 </constructor>
</param>
<param name="StateManager.EntryDataPathName" type="String" generated="false" internal="true">path to data file (eg./tmp/datafile.data)</param>
<param name="StateManager.EntryDataFile" type="Object" generated="false" internal="false" shared="false">
 <class value="java.io.File" />
 <constructor>
  <arguments>
```

```
  <arg order="1" name="pathname" param="StateManager.EntryDataPathName"/>
 </arguments>
</constructor>
</param>
<param name="StateManager.StateStoreInfo" type="Object" generated="false" internal="false" shared="false">
 <class value="gr.uoa.di.madgik.state.store.StateStoreInfo" />
 <constructor>
  <arguments>
   <arg order="1" name="EntryRegistryFile" param="StateManager.EntryRegistryFile"/>
   <arg order="2" name="EntryDataFile" param="StateManager.EntryDataFile"/>
  </arguments>
 </constructor>
</param>
```

The above configuration sets the character set name that will be used for character encoding and decoding and make it accessible under the key EncodingCharset and it will also create a single instance of the StateStoreInfo class that will not be shared across different clients so that it cannot be changed by a different client. This storage information class contains the files that will be used to persist data. If the configuration changes while there are already some entries stored, these entries will not be available after the next reinitialization. The third parametrization value that is defined is whether or not the *StateManager* should, upon initialization, perform an internal cleanup of its repositories.

Store entries and updates

For each entry a client submits to the *StateManager*, a metadata entry is created. This metadata entry contains the following elements:

- Key – When a client registers something to the *StateManager*, he also provides the key by which he wants his object to be identified. This key is also used by the *StateManager* to uniquely identify this object internally. This means that if two clients are using the same key, this entries will be treated as if they were one and the same from the State Store utility.
- Active flag – As a client can store and retrieve an object based on its key, he can also remove it from the State Store. This is done by setting the entry as inactive.
- Data Offset – The metadata entry is also the placeholder for storing the position of the actual data it represents in the data file that it has been persisted. So the starting and ending offset of the data in the data file is stored in the entry.
- Registry Offset – Additionally to the data, the metadata entries are also persisted so that on reinitialization the metadata entries can be reconstructed but also updated during the lifetime of the entry. For this reason the exact position of the metadata entry in the registry file is also kept.
- Type – A number of different types of entries can be stored and retrieved by a client. Depending on their type the *StateManager* treats them differently.
    - Alphanumeric – This represent a simple String that can be stored. To store the string as a stream of bytes, a configurable character set is used.
    - Byte Array – This represent an array of bytes that the client code provides and is stored as is in the data file.
    - File – This represent a file accessible form the local filesystem that the State Store will read through and persist its full content.
    - Serializable – This represents an object that is serializable as the java.io.Serializable interface dictates and can be serialized and deserialized through the ObjectInputStream and ObjectOutputStream java classes.

- ISerializable – This represents an object that implements a custom interface that defines serialization and deserialization methods so that the class can define its own needed serialization payload.

The *StateManager* defines different *put* methods for each of the described entries as well as typed *get* convenience methods so that the client code does not have to perform any type casting or additional serialization or deserialization operations. One of the additions that will be delivered in later versions is to add compression capabilities for the data the clients store.

During the lifetime of a State Store, records become updated, deleted and inserted. Since the registry as well as the data files are used in a sequential fashion to store new entries, and updates of the data payload cannot be performed over the discarded older data to simplify data allocation policies and speedup buffered sequential reads, both files might end up fragmented having large spaces of unallocated spans. Files tend to grow and the State Store repositories are no different.

For this reason the *StateManager* exposes a method that cleans up the data and registry files discarding entries that have been deleted, data that have been updated etc, compacting both files leaving only the valid entries that are left. Since this process entails reading both files fully and writing the valid payload again, it might impose a non negligible disk overhead depending on the amount of data stored in the repository. Since during this process the in-memory metadata data structure is not in-sync with the repository files, the *StateManager* does not allow any client to make any kind of operation.

This operation can currently be initiated by any client code, or using the Configuration entry *StateManager.CleanUpOnInit* which, if set, will dictate the *StateManager* to perform the clean up upon initialization. In later versions more sophisticated options will become available such as statistics of unneeded records, resources availability, client activity etc.

Channel

Another utility offered by the *MadgikCommons* library is the *Event Channeling mechanism*. In a distributed environment, a lot of scenarios can make use of a mechanism that offers them the possibility to open a communication channel among two or more entities to publish information that will travel to all of the channel partners.

The *Event Channeling mechanism* offers this functionality through a distributed event mechanism. A single entity initiates a channel and publishes its existence. This point acts as a sink for the events the other entities emit and as a broker to reproduce the incoming events to the remaining entities. Other entities that receive the identifying object pointing to the created channel connect to it and can from then on receive events that the rest of the entities emit, but also emit their own events.

Channel Events

As the *Event Channeling mechanism* bases its functionality on the brokerage of events, a fundamental concept of the mechanism is the *ChannelEvent*. The events declared here are the ones that define the flow of control governing the lifetime of a channel, its creation, brokerage, synchronization and destruction. Each record pool has associated a *ChannelState* class in which the events that the channel can emit are defined and through which an interested party can register for the event. The *ChannelState* keeps the events in a named collection through which a specific event can be requested.

The event model used is a derivative of the Observer – Observable pattern. Each one of the events declared in the package and stored in the *ChannelState* event collection extends java Observable. Observers can register for state updates of the specific instance of the event. This of course means that there is a single point of update for each event which automatically raises a lot of issues of concurrency, synchronization and race conditions. So the framework uses the Observables for Observers to register with, but the actual state update is not carried through the Observabe the Observers have registered with, but rather as an instance of the same event that is passed as an argument in the respective update call. That way the originally stored Observable instances in the *ChannelState* act as the registration point while a new instance of the Observable that is created by the emitting party is the one that actually carries the notification data.

All events extend the *ChannelStateEvent* which in turns implements the Observable interface and the *ISerializable* . The later is used during transport to serialize and deserialize an event and is explained in later sections. Events containing payload information that the client wants to emit to registered entities, extend the *ChannelPayloadStateEvent* which in turn extend the base *ChannelStateEvent*. The events defined and are the bases of the framework are the following:

- *DisposeChannelEvent* – This event is emitted when the channel is no longer needed and signals the disposal of the channel and all associated state. Since more than one entities, and not limited to two can be part of a single channel, any of these entities can cause the *DisposeChannelEvent*.
- *BytePayloadChannelEvent* – This event can be emitted by any of the channel's nozzles and be carried to all the rest of the participating nozzles. It carries with it a payload of a byte array that can be set on the emitting point.
- *StringPayloadChannelEvent* – This event can be emitted by any of the channel's nozzles and be carried to all the rest of the participating nozzles. It carries with it a payload of a string that can be set on the emitting point.
- *ObjectPayloadChannelEvent* – This event can be emitted by any of the channel's nozzles and be carried to all the rest of the participating nozzles. It carries with it a payload of an object that

can be set on the emitting point. The object payload must implement a needed *ISerializable* interface that declare how the object can be serialized and deserialized in case over the wire transportation is needed.

Proxies and Registry

The two main actors in an *Event Channeling mechanism* scenario is the entity that creates an event sync / broker (*inlet nozzle*) and a number of entities that can register to this and consume events other entities emit but also emit their own events (*outlet nozzle*). The *inlet* creates a channel that the *outlet* needs to access. The two main cases one may distinguish between in this scenario is an *outlet* that is collocated with the *inlet* in the same host and the same VM in which case they share a common address space. The second case is that the two actors do not share the address space. This may mean that the *inlet* and *outlet* are running in a different VM either in the same or in a completely different host. The *inlet* and *outlet* should not be forced to make this distinction in their code. The *Event Channeling mechanism* frees them from handling the two cases differently and allows them to have the same behavior in their code regardless of the other party location. This is achieved through the proxy concept.

A proxy is in fact a mediator between the two actors. Different types of proxies can handle different type of communication, can be initialized with different synchronization protocols, and use different underlying transport technologies. All this are known to the client only at a declarative level in case he may want to have more control over the mechanics of the framework. Different implementations if the proxy semantics can be provided by implementing the *IChannelProxy* interface. Every proxy implementation will still need to provide a marshalable identification mechanism which can be send over to an *outlet* side which will in turn use it to initialize a new proxy instance that will mediate on his side the channel synchronization. This object is an implementation of the *IChannelLocator* interface. Implementations of this interface are capable to identify uniquely a channel. They must also hold enough information to contact the *inlet* side proxy that mediates the instantiating side of the mechanism. This implies information on the protocol used, and any protocol specific information needed by the *outlet* side proxy.

In any case of either collocated and remotely located *inlets* and *outlets*, there is always the case where a channel needs to be identified within the address space of the VM that it was created. To do so, since the identification token needs to be serializable and movable to a remote location a java reference is not adequate. There needs to be a different construct that will identify uniquely the channel and through it the channel itself can be retrievable. This construct in the *Event Channeling mechanism* is the *ChannelRegistry* class. The *ChannelRegistry* is a static class unique within the context of a VM in which channels can registered, assigned within a unique id and discoverable through it. Whenever a *inlet* wants to make the channel it is creating available for consumption trough a proxy, it registers the channel and the proxy through which the channel will be available with the *ChannelRegistry* class. The registration procedure produces a registry key consisting of a UUID. The channel and associated proxy is stored in the registry for future reference through the registry key and the produced registry key is provided back to the registration procedure caller to use it to produce the locator that will be able to identify the channel through it. The registry construct needs also be registered to some of the channel events so that it will be able to perform its own cleanup once the channel is disposed.

The *ChannelRegistry* is also the place where additional information on the function of each channel it stores is kept. This information includes the number of *outlets* that can be connected to a single channel. Since every time an outlet needs to get connected to a channel it must pass through the

ChannelRegistry, this is also the policy enforcing point on the number of *outlets* that can be connected to a single channel depending on the initialization configuration. Additionally, the *ChannelRegistry* keeps track of the method each *outlet* uses to connect to each channel as well as their identity. This way proper cleanup can be performed, but also in cases of broadcasting like behavior, the whole set of *outlets* connected to a single *inlet* can be discovered.

Local Proxy

A common case in a dynamically distributed environment scenario is the one in which the entities that need to communicate are collocated, running within the boundaries of the same VM, possibly in different threads if their execution is concurrent or even serially one after the other. In such a situation it is obvious that for performance and for resource utilization reasons one would prefer to be able to access directly through the shared memory address space the events the *inlet* is emitting or brokering within the *outlet*.

The *Event Channeling mechanism* design and operation facilitates its usage as a shared event registration and brokerage point. The synchronization of *inlet* and *outlet* parties through events makes transparent for both actors the actual location of their counterpart. In this environment the only thing missing to enable optimized local consumption of a channel is a way to pass to the *outlet* a reference to the channel handled by the *inlet*. To do so in a manner that would hide from the two actors the actual location of each other and not force them to apply any kind of logic to distinguish between the two cases, the mechanism's registration and mediation classes comes in.

Once the *inlet* is ready to start its function as an event sync and brokerage point, it will register its channel with the *ChannelRegistry* class as described in previous sections. For the registration step an *IChannelProxy* implementing instance must be provided. Should the *outlets* that will need to contact the created channel need to be restricted to being collocated with the *inlet* (this knowledge is assumed to exist somewhere in the system, quite possibly to the component orchestrating the execution) an instance of the *LocalChannelProxy* class will be provided for the registration step. During the registration process the created channel will be assigned to the *LocalChannelProxy* class. From then on the proxy can be queried to produce an *IChannelLocator* instance. The *LocalChannelProxy* class will produce an instance of the *LocalChannelLocator* class. This locator can me serialized and deserialized to produce a new *LocalChannelLocator* instance that can still identify the referenced channel through the *ChannelRegistryKey* that was assigned to the channel and set to the *LocalChannelProxy*. This locator can in turn be used to create a new *LocalChannelProxy* instance and used to instantiate a new *outlet* that can connect to *inlet's* channel. The channel used will be a reference to the one produced since the *LocalChannelProxy* can lookup the *ChannelRegistry* to retrieve the stored channel associated with the *LocalChannelLocator* stored registry key.

TCPServer Proxy

The general case in a dynamically distributed environment scenario is to be unaware of the location of the two actors or specifically know that they are located in different hosts or in general different VMs. This means that no in memory reference can be shared. Connected nozzles must communicate externally to be able to synchronize their operation and exchange events.

As has already been described in the *LocalChannelProxy* synchronization scenario the *outlets* and *inlets* of a channel are already able to synchronize based on events emitted through the channel. In the protocol section we will illustrate how the events and data that are passed through a channel can be

transferred and mirrored to a remote copy of the channel. The only thing left to enable remote production and consumption is a technology to transfer the protocol packets. Multiple protocols can be used and in this section the case of a TCP connection use case will be illustrated. Since the *inlet* and *outlets* need not be aware of any of the underlying technology details, the proxy and registration classes of the framework are the ones managing all needed technology specific details.

As in the case of the *LocalChannelProxy* the *inlet* at some point needs to share his event aggregation channel with a number of *outlets*. He will again go through the registration procedure but this time instead of a *LocalChannelProxy* proxy instance, a *TCPServerChannelProxy* instance is passed to the registration method. The first time a producer creates an instance of the *TCPServerChannelProxy* within the boundaries of a VM, a *TCPTransportServer* instance is created following the singleton pattern. From then on any instance of the *TCPServerChannelProxy* is directed to this server to manage its communication needs. The port the server listens to is configured through the already available configuration class of the proxy instance and after initialized, all traffic will pass through this port.

Once the registration process is completed by the *inlet,* the *inlet*'s side proxy, transport mechanism and protocol are all ready to accept incoming calls. The proxy is in a position to create a valid *IChannelLocator* instance which for the *TCPServerChannelProxy* will be a *TCPChannelLocator*. The *TCPChannelLocator* contains the referenced channel *RegistryKey*, the hostname the producer is running on and the port the server is listening to. This locator can be serialized and deserialized into a new *TCPChannelLocator* that can be used to initialize an *outlet* to access the remote channel through a newly initialized trasnsporting protocol mechanism created from the *TCPServerChannelProxy* of the *outlet*.

The server uses the *ChannelRegistry* to keep and update a mapping between served channels and open sockets. Upon request it can provide interested parties with the sockets that are associated with the channel they are mediating for through the *ChannelRegistry*. Upon disposal of the channel, the internal state of the *ChannelRegistry* connected to the specific channel is also purged along with the connected sockets.

To be able to create such a mapping between the channel and the specific socket, the server bases its functionality to the *outlet* send UUID of the channel it needs to connect to as provided through the locator instance it is initialized with. This information is always send out as clear text along with an identifier that will uniquely identify the connecting nozzle.

Furthermore a vulnerability this implementation has is the way the server expects to retrieve the referenced channel UUID and nozzle identifier to map the opened socket. After opening a socket, the server will block waiting for the consumer to send its channel UUID and nozzle identifier. Until the information is provided, no other *outlet* will be accepted by the server. This leaves open ground for a denial of service attack but was chosen not to be handled by a separate thread spawned to save resources.

Synchronization protocol

Whenever an *inlet* and a number of *outlets* share a channel there needs to be some synchronization between them both to ensure access integrity but also to enable remote consumption. When the *inlet* and *outlets* are collocated, the synchronization is done without mediation directly through the events exposed by the shared channel. When the communication is mediated there is a need for a synchronization protocol that will handle the forwarding of locally produced events and their interpretation as well as the marshaling and unmarshaling of events and their payload.

This is the work of the synchronization protocol implementing classes. Specifically there are

two classes that participate, the *InletProtocol* and *OutletProtocol*, that act on behalf of the *inlet* and *outlet* respectively.

Outlet Protocol

First the protocol needs to create a local copy of a channel the *inlet* is serving. It contacts the *InletProtocol* sending the channel identifier and the nozzle identifier of the *outlet* it acts on behalf of. After this the thread enters the protocol loop that is ended once the the local or remote channel is declared as disposed. The three steps of the protocol loop are

1. Send any locally produced events that have been produced directly by the mediated nozzle.
2. Receive any events that the inlet has to send and forward them to the local channel unless the received event was originally emitted by the mediated nozzle.
3. Repeat either when a configurable predefined period has elapsed or the outlet has produced some event.

Inlet Protocol

The protocol acts on behalf of the *inlet* nozzle and is responsible of aggregating all connected *outlet* nozzle events, as well as forwarding aggregated events to all connected *outlet* nozzles as well as the *inlet* channel. The protocol thread once created, enters a protocol loop whose steps are the following:

1. Consults the *ChannelRegistry* to find the connected *outlets* to the channel it mediates for.
2. Waits for all connected *outlet* nozzles to be ready to send their produced events
3. Receives all the events each *outlet* nozzle has to send and for each event, if it was not originally emitted by the *inlet*, it forwards it to the local channel.
4. Emits to each *outlet* nozzle the events that were produced in the local channel since the last iteration as long as the event was not originally emitted by the current *outlet* nozzle.

The reason the protocol waits for all connected *outlet* nozzles to be ready before it starts its protocol iteration is to simplify the procedure as otherwise it would need to keep a detailed mapping of all events it has send to each connected *outlet*. This in cases of long interactions and selective big delays could mean a possibly very big detailed list that would consume a lot of resource. On the other hand the current approach could cause a delay on healthy outlet nozzles because of a single outlet nozzle with bad connectivity. Alternatives to the problem will be investigated in later versions.