C++ Multicast Distributor

**2012-02-13**

Contents

[Overview 2](#_Toc315781111)

[Jump Start 3](#_Toc315781112)

[Overall Design 4](#_Toc315781113)

[Objects 4](#_Toc315781114)

[Distributor 5](#_Toc315781115)

[DistributorConnectionInterface 6](#_Toc315781116)

[DistributorPublisherInterface 6](#_Toc315781117)

[DistributorSubscriberInterface 6](#_Toc315781118)

[Configuration 7](#_Toc315781119)

[DistributorApplicationConfiguration 7](#_Toc315781120)

[DistributorConnectionConfiguration 8](#_Toc315781121)

[Callbacks 12](#_Toc315781122)

[Event Callback Interface 12](#_Toc315781123)

[Update Callback Events 13](#_Toc315781124)

[Nagging Distributor Connections 14](#_Toc315781125)

[Subject Names 15](#_Toc315781126)

[Release 16](#_Toc315781127)

[Some Few Things That May Could Be Good To Know 17](#_Toc315781128)

[DistributorInitialize() 17](#_Toc315781129)

[Kernel Buffers 17](#_Toc315781130)

[Contact 18](#_Toc315781131)

# Overview

The Distributor utility is a package implementing a publish /subscribe mechanism. The package provides the following features

* True *one-to-many* distributor i.e. uses IP multicast as transport mechanism.
* Guarantee delivery, implements recovery, guarantying that messages are not lost or duplicated (error free transport, not store-and-forward).
* Highly efficient message distribution with focus on latency and throughput.
* By using IP multicast groups and subject filtering application can select and subscribe to information just relevant for the application.

# Jump Start

* Unpack the kit. The kit is provided as a compress tar file. To uncompress and untar the distributor, issue the following commands;
  + $ gzip –d distributorCpp-<version>.tar.gz
  + $ tar –xvf distributorCpp-<version>.tar
* The distributor CPP related files are the found in the *base director* ./distributor/cpp/
* Take a look in the file ./publisher/publisher.cpp or ./subscriber/subscriber.cpp. Go to the bottom of the file main() routine and you most likely will figure out how it works.

int main( int argc, char \*\*argv )

{

parseParameters( argc, argv );

DistributorInitialize(); //initial the distributor environment

DistributorEventHandler \*tDistRcvEventHandler =   
 new DistributorEventHandler("subscriber");

/\* Create a distributor instance \*

DistributorUpdateHandler \*tDistUpdateHandler = new DistributorUpdateHandler();

DistributorApplicationConfiguration tApplConfig("SubscribeTest");

tApplConfig.setLogFlags( mParamLogFlags );

/\* Create a distributor configuration, connection and subscriber \*/

DistributorConfiguration tConnConfiguration( mParamMcAddr, mParamMcPort);

tConnConfiguration.setSegmentSize( mParamSegmentSize );

tConnConfiguration.setIpBufferSize( mParamIpBufferSize );

tConnConfiguration.setFakeRcvErrorRate( mParamFakeRcvError );

tConnConfiguration.setEthDevice(mParamEthDevice);

Distributor \*tDistributor;

DistributorConnectionInterface \*tDistConn;

DistributorSubscriberInterface \*tSubscriber;

try {

tDistributor = new Distributor(tApplConfig);

tDistConn = tDistributor->creatConnection(tConnConfiguration);

tSubscriber = tDistributor->createSubscriber(tDistConn,   
 tDistRcvEventHandler,  
 tDistUpdateHandler);

tSubscriber->addSubscription("/foo/bar/frotz", NULL);

} catch (DistributorException e) {

std::cout << "Distributor Exception: " << e.getMessage() << std::endl;

exit(666);

}

while( true ) {

sleep( 10 ); // wait for updates to be delivered via the callback

}

}

# Overall Design

* The Distributor is a true one-to-many mechanism. When publishing updates it sent once and is received by one or more subscribers that have declared an interest in the information.
* The distributor uses IP multicast (RCF 1112) when publishing update messages. The IP multicast transport is not a guarantee transport, messages can get lost. The distributor implementing a recovery schema guarantying published messages not being lost or duplicated.
* Negative acknowledgment is used to signal detection of lost messages. Receivers do not acknowledge received messages as long as messages being received in sequence. In case a receiver discovers a loss of messages a retransmission request is published to the publisher requesting retransmission of missed messages. Retransmission and recover takes place over the IP multicast channel used for publishing. All recovery logic is internal to the distributor package and hidden for the applications using the distributor utility.
* Publishers keep sent messages in a retransmission cache and could serve retransmission requests from receivers as long messages are in the retransmission cache. The retransmission cache is a FIFO cache with a limited size (configurable).
* Subscribers will only receive updates that they subscribe to. Each message published is associated with a subject. A subject is a string having a name hierarchy like “/foo/bar/fie”. Subscribers can subscribe to specific subjects or a group of subjects using wildcard subscriptions.
* An application may connect up to one or more physical IP multicast groups. Publishers and subscribers that would like to share information must be connected to the same IP multicast groups i.e. IP Class D address and IP port.
* The Distributor utility is provided as a shareable library with and API

# Objects

* The Distributor functionality is provided as API.
* The API is a C++ interface with the following classes
  + **Distributor**, main class for accessing the Distributor interface.
  + **DistributorApplicationConfiguration**, common global settings for the Distributor
  + **DistributorConnectionConnfiguration**, settings for a Distributor connection.
  + **DistributorConnectionInterface,** is a transport connection associated with a IP multicast group i.e. IP class D address and IP UDP port.
  + **DistributorPublisherInterface,** a publisher interface allowing applications to publish data on a distributor connection.
  + **DistributorSubscriberIntgerface,** a subscriber interface allowing applications to subscribe to data published on distributor connection

## Distributor

The Distributor class provides a main handle to the distributor utility. Typically an application creates one instance of a *Distributor* object. The Distributor provides the following interface methods. For detailed definitions see “./include/Distributor.h”

**Distributor**( const DistributorApplicationConfiguration &pConfiguration );

**Distributor**( const DistributorApplicationConfiguration \*pConfiguration );

* Constructor for creating a *Distributor* instance
  + pConfiguration, distributor application configuration parameters

DistributorConnectionInterface \***creatConnection**(

const DistributorConfiguration &pConfiguration);

DistributorConnectionInterface \***creatConnection**(

const DistributorConfiguration \*pConfiguration);

* Method for creating a distributor connection instance (i.e. physical transport)
  + pConfiguration, parameter object defining the setup for the connection.

void **closeConnection**(DistributorConnectionInterface \*\*pDistributorConnectionInterface);

* Method for close down and delete a distributor connection. Deleting a distributor connection instance will have the same effect as calling the closeConnection method.
  + pDistributorConnectionInterface, pointer to an address refereeing to a distributor connection object

DistributorPublisherInterface \***createPublisher**(

DistributorConnectionInterface \*pDistributorConnectionInterface,

DistributorEventCallbackInterface \*pEventCallback)

throw (DistributorException);

* Method for creating a publisher instance.
* pDistributorConnectionInterface, distributor connection that the publisher will send update messages on when publishing data.
* pEventCallback, event callback interface receiving application events

DistributorSubscriberInterface \***createSubscriber**(

DistributorConnectionInterface \*pDistributorConnectionInterface,

DistributorEventCallbackInterface \*pEventCallback,

DistributorUpdateCallbackInterface \*pUpdateCallback)

throw (DistributorException);

* Method for creating a subscriber instance.
* pDistributorConnectionInterface, distributor connection that the subscriber will read and receive updates on.
* pEventCallback, event callback interface receiving application events.
* pUpdateCallback, interface invoked when updates are received and passed back t the application

## DistributorConnectionInterface

Is just a handle and do not have any methods others than a *toString* method.

## DistributorPublisherInterface

The distributor publisher interface is used by applications publishing information on a distributor connection. A publisher interface is obtained by calling the *Distributor* *createPublisher* method. The interface provides the access methods found below. For detailed information about the publisher interface see “./include/DistributorPublisherInterface.h”

uint32\_t **publish**( const std::string &pSubjectName,

const void \*pData, const int32\_t pDataLength )

throw (DistributorException)

* Method for publish updates to subscribers
* pSubjectName, subject name associated with data being published.
* pData, address to the update being published.
* pUpdateCallback, length of the update data being published.

## DistributorSubscriberInterface

The distributor subscriber interface is used by applications subscribing to information published on a distributor connection. A subscriber interface is obtained when calling the *Distributor* *createSubscriber* method. The interface provides the access methods found below. For more detailed information about the subscriber interface see “./include/DistributorSubscriberInterface.h”

void \* **addSubscription**( const std::string &pSubjectName, void \* pCallbackParameter)   
 throw (DistributorException);

* Method add subscription
* return handle for the subscription, the handled should be used when a subscription is to be canceled i.e. when the method *cancelSubscription* is invoked.
* pSubjectName, subject to subscribe for. The subject may contain wildcards. For more information about wildcard subscriptions see “Subject Name” below.
* pCallbackParameter, parameter to be passed to the callback method when a matching update is received.

void **removeSubscription**(void \* pHandle)   
 throw (DistributorException);

# Configuration

There are not many class methods an application needs to use in order to publish or subscribe to data. The interface is very small and simple.

There is however two classes used for configure the behavior of the utility, *DistributorApplicationConfiguration* and *DistributorConnectionConnfiguration* . The configuration classes are instantiated with default values when being created. The belief is that the default values should be applicable in most cases. Below the set access methods for the two configuration objects are listed.

## DistributorApplicationConfiguration

**DistributorApplicationConfiguration**(const std::string &pApplicationName );

* Constructor creating and instantiating a *DistributorApplicationConfiguration*. Later used to as in parameter when creating a *Distributor* object. The application configuration class is just used to collect configuration parameters used by the Distributor.
* pApplicationName, name identifying the distributor application.

void **setLogFlags**( int32\_t pLogFlags );

* Enable / disable logging. The log flag argument is mask defining the logging being enabled. Log events are written to stdout and the distributor log file if being enabled.
* pLogFlag, mask defining the application and connection logging being enabled.

The following log flags being defined

|  |  |  |
| --- | --- | --- |
| Symbol | Value | Logging |
| LOG\_ERROR\_EVENTS | 1 | General warnings and error |
| LOG\_CONNECTION\_EVENTS | 2 | When a application creates or delete distributor connections |
| LOG\_RMTDB\_EVENTS | 4 | When a remote distributor application creates or deletes remote connections |
| LOG\_RETRANSMISSION\_EVENTS | 8 | Logging and tracing of retransmission requests and responses |
| LOG\_SUBSCRIPTION\_EVENTS | 16 | Logging of subscriber add and remove subscription requests. |
| LOG\_STATISTIC\_EVENTS | 32 | Periodic logging of distributor statistical information. |
| LOG\_SEGMENTS\_EVENTS | 64 | Logging and tracing of sent and received segments (headers) |
| LOG\_DATA\_PROTOCOL\_RCV | 128 | Logging and tracing of distributor protocol messages received. |
| LOG\_DATA\_PROTOCOL\_XTA | 256 | Logging and tracing of distributor protocol messages sent. |
| LOG\_RETRANSMISSION\_CACHE | 512 | Logging of cache statistics such as size, messages in cache etc. |

*By default the following flags are enabled  
LOG\_ERROR\_EVENTS, LOG\_CONNECTION\_EVENTS, LOG\_RETRANSMISSION\_EVENTS, LOG\_RMTDB\_EVENTS*

void **disableLogging**(**);**

* Normally the distributor will create a log in the working directory file to which events will be written. The creation of the log file and logging of events can be disabled if invoking this method before the Distributor object is created.

# DistributorConnectionConfiguration

The distributor configuration is class holding configuration attributes used by a distribution connection. All attributes in a *DistributorConnectionConfiguration* object is instantiated with default values when the object is created. These setting are sufficient for most applications. Modification of configuration attributes must be done before they are passed to the ***Distributor::createConnection*** attributes changed after the connection as been created will have no effect.

**DistributorConnectionConfiguration**( const std::string &pMca, int32\_t pMcaSourcePort);

* Constructor creating a distributor configuration object.
  + pMca, IP multicast group being used when sending and receiving data. The IP address is a class D address in the range 224.0.0.0 to 239.255.255.255.   
    (<http://en.wikipedia.org/wiki/Multicast_address>)
  + pMcaSourcePort, IP UDP port used .

void setStatisticFilename(const std::string &pFilename);

* Sets the name to which the distributor statistics should be written to. The default name is <application-name>-Statistics-<ip-mc-address>\_<ip-mc-port><timestamp>.log
  + pFilename, filename of the file to which statistic information will be written.

void setMcaNetworkInterface(std::string pInterface);

* Sets the IP interface on which the distributor will send and receive data. Default the IF\_ANY will be used. This will typically be the default interface. If specified it should be the IP interface address of the device to use.
  + pInterface, the ip address of the interface to be used.

void setSenderIdPort(int32\_t pValue);

* Sets a connection sender port. Multiple applications can start the same connection i.e. use same UDP address / UDP port in order for receivers to make a distinction between two different application using the same connection on the same host a connection will use an additional sender-id-port". Normally the distributor allocates this "sender-id-port" implicit when the connection (by binding a server tcp/ip). But for some reason it is possible to set this manually via this routine. Just let it be!
  + pValue , sender id port

void setTTL(int32\_t pValue);

* Set the TTL value to use when multicast data, values are in the range 0-255. Default value is 255.
  + pValue, Time-To-Live value

void setIpBufferSize(int32\_t pBufferSize);

* Set the IP receive (SO\_RCVBUF) and send (SO\_SNDBUF) buffer size for the socket used when sending and receiving IP multicast data.
  + pBufferSize, buffer size in number of bytes, default value 128000

void setHeartbeatInterval(int64\_t pValue);

* Set the heartbeat interval. The distributor will send a heartbeat periodically with the latest sequence number published on the distribution connection in case no data is transmitted.
  + pValue, heartbeat interval in seconds, default value is 3 seconds.

void setHearbeatInterval(int64\_t pValue);

* Set max number of missed heartbeats that an application can accept are accepted before the connection is considered to be lost.
  + pValue, max allowed missed heartbeats, default value 10

void setMaxBandwidth(double pValue);

* Set the max number of bytes / second the distributor can publish on the connection. In case the application tries to publish more data the publishing calls are stalled so that the pusblishrate does not exceeds the max bandwidth.
  + pValue, max bandwidth in MB/sec, default value is 0 i.e. unlimited bandwidth, no checks.

void setRetransmissionServerHoldback(int64\_t pValue);

* Set the server retransmission holdback time (in milliseconds). When a retransmission request is received the missed message will not be re-sent immediately in case holdback value is > 0. In case a message is missed it's not unlikely that more than one receiver is missing the message. By hold back the re-publishing of the missed message multiple retransmission requests can be "served" if they arrive within the holdback interval, this since retransmissions are sent out as an UDP message.
  + pValue, retransmission holdback time in milliseconds, default value 20 ms

void setRetransmissionTimeout(int64\_t pMilliseconds);

* Set the retransmission timeout. Defines the time a a connection will wait for a retransmission to be re-published. If not being served another retransmission are requested, utile the “max-number-of-retransmission” has been exceeded.
  + pValue, retransmission timeout in milliseconds, default value 800 ms

void setRetransmissionRetries(int32\_t pValue);

* Set the maximum number a times a connection will try to recover a missed message i.e. max-number-of-retries.
  + pValue, max number of retransmission retries, default value 10 times.

void setRetransmissionMaxCacheSize(int32\_t pValue);

* Set the maximum size (in bytes) of the retransmission cache. When the upper limit is reached and additional messages are to be added to the cache old messages are removed so that the cache size does not exceed the defined threshold value (i.e. FIFO).
  + pValue, retransmission cache size in bytes, default value 10000000

void setRetransmissionCacheLifeTime(int32\_t pValue);

* Set the time a message is about to be kept in the retransmission cache. Message being kept longer than the cache life time are removed from the cache. Messages are not guaranteed to be kept in the cache for the whole cache life time period. A message may be cleaned out in case the transmission rate is high and the retransmission cache message turnover time is less than the cache life time.
  + pValue, message cache life time in seconds, default value 30 sec.

void setRetransmissionCacheCleanInterval(int64\_t pInterval);

* Set the periodic interval with which the retransmission clean job will run. The clean job will remove all messages in the retransmission cache that have lived more than the retransmission cache life time.
  + pValue, cache clear interval in milliseconds, default value 2000 milliseconds.

void setNaggingWindowInterval(int32\_t pValue);

* Set the nagging window interval. For more information about the nagging monitor functionality see section “Nagging Distributor Connections” below.
  + pValue, nagging window interval in milliseconds , default value 4000. A value of 0 disables the nagging monitor functionality.

void setNaggingCheckInterval(int64\_t pValue);

* Set the nagging check interval. For more information about the nagging monitor functionality see section “Nagging Distributor Connections” below.
  + pValue, nagging check interval in seconds. Default value

void setNaggingMaxRetransmissions(int32\_t pValue);

* Set the max number of retransmissions that could be sent under the nagging check interval without a nagging event should be generated. For more information about the nagging monitor functionality see section “Nagging Distributor Connections” below.
  + pValue, max number of retransmissions.

void setConfigurationInterval(int64\_t pValue);

* Set how frequently distributor configuration messages are sent. Connection configuration messages will notify remote distributor applications about the presents of distributor applications.
  + pValue, send frequency in milliseconds, default value 15000 ms.

void setConfigurationMaxLost(int32\_t pValue);

* Defines how many configuration messages that could be missed or lost before a remote distributor connection is consider being unavailable.
  + pValue, max lost configuration messages.

void setStatisticsLogInterval(int64\_t pInterval);

* Defines how frequently distributor connection statistics are to be written to the statistics log.
  + pInterval, log frequency in milliseconds, default 0 (logging disabled).

void setFakeXtaErrorRate(int32\_t pValue);

* Use for debugging and testing to simulate transmission errors. It defines the frequency with which sent messages will be dropped. The value express %% rate. A value of 10 will result in 10 messages out of 1000 to be dropped randomly.
  + pValue, drop rate in %%

void setFakeRcvErrorRate(int32\_t pValue);

* Use for debugging and testing to simulate transmission errors. It defines the frequency with which received messages will be dropped. The value express %% rate. A value of 10 will result in 10 messages out of 1000 to be dropped randomly.
  + pValue, drop rate in %%

void setFlowRateRecalcInterval( const int64\_t pCheckInterval );

* If outgoing bandwidth restriction has been defined for the distributor. This parameter defines how frequency the used output flow should be recalculated.
  + pValue, recalculation interval in milliseconds. Default value 100 ms.

void setSendHoldbackDelay(int64\_t pHoldbackDelay);

* Normally published updates are sent immediately, resulting in one update one physical package. The send holdback delay defines a sender holdback. The transmission of packets a held back with the time specified by the holdback parameter. In case there are other updates being sent within the same time are inserted in the held back package. This will result in one or more updates being packed into one package. A classic latency vs. throughput issue.
  + pValue, holdback time in milliseconds. Default value 10 ms. A value of 0 will result in update to be sent immediately under all circumstances.

void setSendHoldbackThreshold(int32\_t pHoldbackThreshold);

* Defines when the holdback timer is to be used. The holdback timer will not be inforce unless the publishing update rate exceeds the holdback-threshold value.
  + pValue, sender holdback threshold. Default value 200 updates / sec.

void setSegmentSize(int32\_t pValue);

* The distributor is not bound to any particular message size. Internally the distributor will use a defined package size i.e. segment size. A package may stack one or more updates into one package in case updates are smaller than the package size. Large update > package size are segmented and sent in multiple package. The segment size defines the size of the package to be used internally by the distributor. Note that the segment size must be the same for all distributor application sharing data.
  + pValue, segment size in bytes. Default value 819 2 (max value 64000).

void setLocalAddress(const std::string &pLocalAddress);

* The local address is used to identify the host when interacting with remote distributor hosts. The node address has no other meaning other than identification for information purpose. Normally the host address is retrieved from an Ethernet device selected (defined by the ***setEthDevice*** method). Using the address of the specified Ethernet device is the standard procedure. You can however override the address by invoking the method ***setLocalAddress*** manually.
  + pLocalAddress, local IP address to use for informational identification.

void setEthDevice( const std::string &pString );

* The distributors identifies themselves with the local IP address. The IP address used will be the one defined for the eth device specified by this parameter. It the parameter local-host (*setLocalHost*) is set it will take presidents.
  + pString, Ethernet device name. Default value eth0:

void setSmallSegmentSize(int32\_t pValue);

* The distributor transmit internal messages such as heartbeats, configuration messages etc. These messages are fairly small and do not need to use a full package. Therefor smaller packages are allocated on a need base.
  + pValue, small package size to be allocated. Default value 512 bytes.

# Callbacks

### Event Callback Interface

Distributor application provides an event callback interface when creating subscriber or publisher objects. The event callback interface is declared in the file “./include/DistributorCallbackInterface.h”

void **distributorEventCallback**(const DistributorEvent &pEvent)

The distributor utility will invoke the ***distributorEventCallback*** when there is a significant event to report back to the application. One of the events found below can be reported via the callback interface. The callback events are defined in the files “./include/ Distributor\*Event.h”

* **DistributorNaggingEvent**, the connection used has requested too many retransmissions over a defined time period. The distributor will put itself in a hibernate state and not try to recover lost messages. This event fatal, subscribers and publishers can not trust the state of the connection. The application should disconnect publishers and subscribers using the connection and close the connection.
* **DistributorNakExceptionEvent,** the connection has failed to recover a missed message. The publisher has indicated that the requested message is no longer in the retransmission cache. This is a fatal error and irrecoverable error on the distributor connection. The connection should be closed down.
* **DistributorCommunicationExceptionEvent,** send or a receive failure on the distributor connection socket used. This is a fatal error and irrecoverable error on the distributor connection. The connection should be closed down.
* **DistributorConnectionClosingEvent,** in case an application closes a distributor connection and there is still publishers and subscribers using the connections they are informed that the connection is closing down. It will no longer be possible to publish or receive updates on the connection.
* **DistributorRemoteConnectionEvent,** this event is sent to publishers and subscribers when a new application joins or leave the multicast group used by the distributor connection. This event is informative and has no significance for subscribers or publishers.
* **DistributorRetransmissionFailureEvent**, this event is generated when a missed message cannot be recovered. The number of retransmission retries has been exhausted. This is a fatal error and irrecoverable error on the distributor connection. The connection should be closed down.

### Update Callback Events

When a subscriber instance is created the application provides a update callback interface. The distributor invokes the callback interface when there is an update received matching one of the subscription setup by the subscriber.

*Note! The callback method is invoked in a separate thread.*

void **distributorUpdate**( const std::string &pSubjectName,

const void \*pData,

const int32\_t pDataLength,

void \*pCallbackParameter,

const int32\_t pDeliveryQueueLength);

* Invoked when receiving an update published matching a subscription that the subscriber has setup. Subscriber add subscription by invoking the method ***addSubscription*** in the subscriber interface.
  + pSubjectName, the subject associated with the data being published.
  + pData, user data being published
  + pDataLength, length of the user data
  + pCallbackParameter, user callback parameter being the one used when adding the subscription.
  + pDeliveryQueueLength, number of updates waiting to be passed to the application. A queue indicates that the application is not processing messages in a slower rate than being delivered to the application.

# Nagging Distributor Connections

Each distributor connection monitors itself to examine that it does not generate too many retransmission constantly. Three configurable parameters are used to control the nagging monitor behavior:

* naggingWindowInterval
* naggingCheckInterval
* naggingMaxRetransmissions

If the parameter *naggingWindowInterval* is set to “0” the nagging functionality is disabled.

The nagging monitor will examine if retransmissions have constantly been generated over a time period and if the number of retransmission generated over the period has exceeded a max threshold value. If so a nagging event are generated to the local subscribers and publishers using the connection. Furthermore the connection will stop requesting retransmission and stop processing incoming update messages. This is considered to be a fatal situation and the connection should be closed down by the application.

More in detail the nagging algorithm works as follows:

startTime = 0;  
intervalRetransmission = 0  
totRetransmissions = 0;

// The following code runs periodically with a frequency specified by the parameter, *naggingWindowInterval*

If (intervalRetransmissions > 0) {  
 totRetransmissions += intervalRetransmissions;  
} else {  
 reset startTime, intervalRetransmissions, totRetransmissions;  
}

If (((currentTime – startTime >= naggingCheckInterval) &&   
 (totRetransmission >= naggingMaxRetransmissions)) {  
 generate-nagging-event-to-publishers-and-subscribers();  
} else {  
 reset startTime, intervalRetransmissions, totRetransmissions;  
}

# Subject Names

Subject names are hieratical string like “/foo/bar/fie”. The “/” characters is used to separate levels. A subject name could have an arbitrary number of levels.

The strings “\*” and “…” are used to express wildcard matching

Publisher must publish data with absolute subject names i.e. must not contain the strings “\*” or “…”. Subscribers may use absolute subject or wildcard subject when setting up subscriptions.

Subject names are case sensitive.

Some typical matching rules.

“/foo/bar” will not match “/foo/bar/fie”  
“/foo/\*” will match all subjects with two levels and starting with “/foo”  
“/foo/\*/bar/\*” will match all subjects with four levels where level one is equal with “/foo” level three  
 with “/bar” and level two and four being whatever.

“/foo/bar/…” with match anything with three or more levels starting with “/foo” and “/bar” at level one and two.

# Release

This release is built and tested on LINUX / FEDORA V14.   
(2.6.35.14-97.fc14)

A couple of example programs as been included to exemplify how to use and interact with the distributor. There is essentially two example programs are provider a publisher and subscriber program. There is a console version and GUI version of each program.

The distributor functionality is currently provided as a ***32 bit shareable library***. The library libstdc++ is statically linked into the libdistributor.so in order to avoid compatibility issues with the libstdc++.

When untar the release the following directories are created under the base directory ./distributor/cpp/

|  |  |
| --- | --- |
| ./distributor/include | Contains all h-files for distributor classes needed when writing a application utilizing distributor functionality. |
| ./distributor/publisher | Publisher console example program. There is also a make file and *sh* file allowing the example program to be built and run.   * Makefile – compile and link the publisher binary * publisher.sh - script file for launching the publisher binary |
| ./distributor/subscriber | Subscriber console example program. There is also a make file and *sh* file allowing the example program to be built and run.   * Makefile – compile and link the subscriber binary * subscriberr.sh - script file for launching the subscriber binary |
| ./distributor/guisubscriber | Pre-built publisher program for testing and generating traffic. There is a sh file for configure the environment and run the Xsubscriber example program.   * Xsubscriber.sh - script file for launching the Xsubscriber binary |
| ./distributor/guipublisher | Pre-built subscriber program for testing and processing published information. There is a sh file for configure the environment and run the Xpublisher example program.   * Xpublisher.sh - script file for launching the Xpublisher binary |
| ./distributor/lib | Directory where to find the distributor shareable library libdistributor.so The distributor has dependencies to two other libraries   * rt , linux realtime extension library * pthread, the pthread library   The gui example programs are built with the QT framework (<http://en.wikipedia.org/wiki/Qt_(framework)> In order to run these application they would need the QT libraries (V4). The libraries libQtCore.so and libQtgui.so are also to be found in the lib directory. |

# Some Few Things That May Could Be Good To Know

## DistributorInitialize()

There is one global method *DistributorInitialize*. In most cases it could be a good idea to call this method as early as possible in your application. The Distributor internally use ***time\_settime*** together with the *SIGEV\_SIGNAL* mechanism to generate timer events (for more information see <http://www.kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html> ).

The signal used for signal is (SIGRTMAX - 1) i.e. value 63. When *time\_settime* signal a timer event it is delivered to the process and not a specific thread in the distributor library. In case your application do not block user signal 63 for its main thread and threads being created, the application will trap and abort when receiving the signal. The method *DistributorInitialize* routine will block the signal 63 for the main thread and all sub threads being created by the application. The code below it what is executed when invoking the DistributorInitilialize routine.

int32\_t SIGTIMER = (SIGRTMAX - 1);

sigset\_t tSigToBlock;

sigemptyset(&tSigToBlock);

sigaddset(&tSigToBlock, SIGTIMER);

pthread\_sigmask(SIG\_BLOCK, &tSigToBlock, NULL);

## Kernel Buffers

When using IP multicast received and sent data will be intermediately be copied to kernel buffers. By default the buffer configuration may be a little sparse. In case your application aiming for high message rates it is a good idea to increase the buffer size. If not there is a good chance that you will experience data overruns and retransmissions .

You have to have administrator privileges to change the configuration.

// to read current value

$ sysctl net.core.rmem\_max // display max size for kernel read buffers (default 131071)  
$ sysctl net.core.wmem\_max // display max size for kernel send buffers (default 131071)

// to modify the buffer sizes

$ sysctl –w net.core.rmem\_max =16777216  
$ sysctl –w net.core.wmem\_max =16777216

The values could be changed dynamically but will not be persisted. In case you reboot your system the setting will go back to the default values. By adding the values to the file /etc/sysctl.conf values can be set at boot time.

If you would like to get information about if you have any data overruns on the eth interface you can use the ifconfig command.

$ ifconfig <eth-device> ! Typically eth0, eth1 etc

eth1 Link encap:Ethernet HWaddr 00:50:04:2F:1E:4  
 inet6 addr: fe80::250:4ff:fe2f:1e42/64 Scope:Link  
 UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1  
 **RX packets:23099429 errors:0 dropped:0 overruns:0 frame:0  
 TX packets:927844 errors:0 dropped:0 overruns:0 carrier:0**  
 collisions:0 txqueuelen:1000  
 RX bytes:2638670187 (2.4 GiB) TX bytes:108286946 (103.2 MiB)  
 Interrupt:18 Base address:0xef80

## Running Example Program

In the example directories ./publisher ./subscriber ./guipublisher and ./guisubscriber there are bash script files for starting and running the example programs. When unpacking the release the script will not get correct file protection. In order to change the file protection use the *chmod* command.

$ chmod 755 <scriptfile>.sh

## 

## Contact

If you have questions, suggestions or comments feel free to drop me a mail

[boreas@zpalten.com](mailto:boreas@zpalten.com)

ciao,

Pär