



ATLAS TDAQ Controls and Configuration

Message Transport System High-Level Design

Document Version: 0.4
Document ID: ATLAS-TDAQ-CC-001
Document Date: 4 October 2013
Document Status: Draft

Abstract

The Message Transport System (MTS) provides facilities for the reliable transport, the filtering and the routing of the messages reported with the Error Reporting Service (ERS) by ATLAS TDAQ applications and libraries. This document describes main design principles and high-level architecture of the components of MTS.

Institutes and Authors:

NIPNE Bucharest: M. Caprini

PNPI: A. Kazarov

Table 1 Document Change Record

Title: ATLAS TDAQ C&C Message Transfer System High-level Design			
ID: ATLAS-TDAQ-CC-001			
Version	Issue	Date	Comment
0.3	1	04/10/13	
0.4	1	04/10/13	finalised subscription syntax and IDL

1 Introduction

This document describes main design principles and high-level architecture of the components of the Message Transfer System (MTS). MTS is an evolutionary re-design of the former Message Reporting System (MRS) [1] system, based on the reviewed requirements [2] to the component.

1.1 Scope

MTS is one of online infrastructure services provided by the TDAQ software. It is not used directly by the TDAQ applications and libraries, instead ERS API is used for reporting issues and for subscribing to reported issues. However users need to use MTS subscription syntax in order to subscribe to particular subset of messages.

1.2 Glossary

Message (Issue) *a message reported by ERS. It has a predefined structure and may include other (chained) ERS Issues. Further in the document message means an ERS issue while it is being handled by MTS.*

Sender *a TDAQ component (application, library) reporting an Issue*

Receiver *a TDAQ component subscribed to particular subset of Issues. Term **subscriber** can also be used.*

Subscription *an expression for selection of messages given by a Receiver, or a pair {Receiver, Subscription} stored in MTS. Each Receiver may have a number of subscriptions.*

Worker *main MTS application, responsible for routing messages from senders to receivers*

CORBA *a standard for inter-process communication, used in TDAQ online s/w, and middleware implementation of the standard.*

1.2.1 Acronyms and Abbreviations

TDAQ	Trigger and Data Acquisition
ERS	Error Reporting System
CORBA	Common Object Request Broker Architecture

1.3 References

- [1] MRS Architectural Design and Implementation:
https://twiki.cern.ch/twiki/pub/Atlas/DaqHltMrs/mrs_impl.pdf
- [2] MRS Requirements, revised: https://its.cern.ch/jira/secure/attachment/11401/MRS_URD_v2.pdf
- [3] Inter Process Communication (IPC): <http://atlas-onlsw.web.cern.ch/Atlas-onlsw/components/ipc/Welcome.htm>
- [4] Information Service (IS): <http://atlas-tdaq-monitoring.web.cern.ch/atlas-tdaq-monitoring/IS/Welcome.htm>
- [5] EBNF notation (<http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf>)

2 Architectural goals, assumptions, dependencies

2.1 Assumptions

MTS shall handle up to $O(10^4)$ senders and $O(100)$ subscribers. There may be conditions when a number of senders become very verbose and start reporting messages with high rates, e.g. with rates of L2 trigger from L2 or ROS nodes. It is assumed that such potentially verbose senders will configure ERS suppression (throttling) streams, and though the total number of such senders may be high, the condition of message reporting with such high rate is considered as not normal and may last not more than few minutes.

A receiver which can not cope with the incoming rate may start to loose messages after some period of time or when MTS buffers gets full.

In normal conditions, the delivery latency (total time spent in MTS for a message to be delivered from sender to receiver) for a message shall not exceed $2 \cdot t_{\text{ping}} + C$, however in high-load conditions (see above), there may be buffering and queuing of messages such that messages may be reported with additional delays to receivers.

It is assumed that MTS system is part of the common TDAQ partition infrastructure, so normally it is started and configured before its clients (senders and receivers). However, architecture includes scenarios when some parts (e.g. workers) are restarted (e.g. on backup nodes), or when new workers are added to the cloud.

2.2 Main design goals

Reliability: a message must be delivered in case a part of MTS infrastructure (e.g. a worker node or worker process) fails or goes down¹. There shall not be single point of failure in the architecture. In case of a single component (application) implementing a critical functionality, it must be restartable from a backup (file).

Also, MTS shall not cause blocking of user threads on the sender side (e.g. in case of unavailability of parts of MTS infrastructure).

¹ Messages which were kept in workers memory will be lost in case crash of a worker. It is possible to avoid this by adding redundancy to the delivery path, but there is no such requirement.

Scalability: MTS architecture shall be scalable with respect to the number of senders and receivers, providing more throughput by adding more working elements to the system and by utilising all available CPU cores on each node.

Fairness: MTS shall guarantee “fairness” of receivers notification mechanism, i.e. every receiver always get required quantum of MTS resources (worker threads, queues etc. needed to for task of notification) in conditions when some receivers are heavily loaded by the messages (such that messages are buffered and may be delivered with some delays), such that busy receivers do not block delivery of messages to less busy other receivers.

Simplicity: Reuse as much as possible common and standard solutions and existing software, minimise as possible the number of components, interfaces, design complexity.

2.3 External dependencies

MTS uses Inter Process Communication (IPC) [3] naming service for publishing and sharing information about active worker nodes

MTS uses IS (Information Sharing, [4]) system to share information about subscribers and to synchronise it to all workers. IS also used to publish MTS statistical and monitoring data.

MTS uses C++ and Java implementations of CORBA middle-ware as a cross-language and cross-platform transport communication layer over TCP/IP.

3 Use-Case View

See Section 2.3 of the MRS Requirements document [1].

4 Overview of the architecture and functionality of main components

4.1 Communication model

MTS implements a *publish-subscribe-notify* communication pattern with *content-based* message filtering (see http://en.wikipedia.org/wiki/Messaging_pattern, <http://en.wikipedia.org/wiki/Publish/subscribe>), where senders and receivers are linked indirectly: senders send out messages not directly to receivers, but publishes messages to MTS actors (workers) and receivers subscribe in MTS system for particular types of messages and then asynchronously get notified by MTS as soon as a message matching subscription criteria arrives to MTS.

// Figure?

4.2 Architecture overview (component view)

MTS infrastructure consist of a cloud (number of $O(1)$) of identical workers, which task to share and balance the load and to provide fault-tolerant service, in case of failure of a particular worker. There is also a single manager node, performing some of the central tasks (see below).

Details of TCP/IP network communication between MTS actors (senders, workers, receivers) is hidden by use of CORBA protocol. Each actor defines and implements IDL interfaces, providing main functionality. IDL methods are called remotely via CORBA by other actors.

4.2.1 Sender

Sender is a user application which uses ERS configuration and loads MTS client (sender) library which implements `ers::OutputStream`. The library receives ERS Issues from user code, converts it to corresponding IDL structure and sends it as CORBA messages to a worker using its *report* IDL interface. A worker is (randomly) selected from a list of workers registered in IPC naming service and in case the selected worker is not responding², another worker is selected: it is assumed that a pool of workers is started for the sake of redundancy and there will be always a reachable worker available.

4.2.2 Worker

Worker is a main building element of the MTS infrastructure. It provides functionality of accepting messages from sender, filtering (selecting) messages according to known subscription criteria and notifying the receivers. To guarantee the fairness of notification in case of a heavily loaded (or a slow) receiver, a dedicated delivery queue is maintained for each receiver and a pool of worker notification threads is handling all queues in a fair manner, guaranteeing minimum latency delivery for non-slow receivers.

4.2.3 Receiver

Receiver is a user application which implements `ers::Receiver` abstract interface and registers it with named `ers::InputStream` 'mts', implemented by Receiver part of client MTS library. It uses MTS “subscribe” interface to subscribe to particular messages on all workers and provides receiving “notify” IDL interface, which is called by workers. Then receiver restores original ERS Issue object and passes it to user-defined `ers::Receiver::receive()` callback function.

4.2.4 Manager

There are two central management functionalities:

1) sharing list of available workers

This is implemented by utilising the IPC naming service where each worker is registered with a unique ID and sender components can access the list of active workers at any moment. To minimise number of connections to each worker, a worker is selected once by the sender randomly from a list of available workers and then it is used to send all messages. Given the big number of sender applications, all workers will share the load. In case of non-responding or misbehaving worker, sender (randomly) selects another worker from the list. Also, senders may preventively disconnect from a worker (e.g. after some timeout or a number of sent messages, or by receiving some performance metrics from current worker as a result of “*report*” IDL method) and choose another worker – just to improve load balancing. It is assumed that new workers can be dynamically added when system is already running and configured.

2) sharing and synchronising list of active receivers (subscriptions)

Sharing the list of active subscriptions between MTS actors (workers) is needed in the following scenarios, which may be invoked in parallel:

- “*subscribe/unsubscribe*” IDL methods are called by receivers: the subscription data needs to be synchronised (distributed) to all workers, keeping them in sync
- Unsubscription request coming from a worker which decided to disconnect a bad or dead receiver
- A worker is restarted or new worker is added, so it needs to be synced with the other workers and get the same subscriptions list

² In this unlikely case, the sender may be blocked by a timeout required to detect a bad worker, which can be configured to a reasonably low value like 1sec.

This functionality can be implemented as a dedicated MTS manager application, implementing IDL methods “*subscribe*”, “*unsubscribe*” and “*get_subscriptions*”. However this model of sharing the information fits very well with functionality provided by standard IS service, so it looks reasonable to utilise this component and to keep list of active subscriptions on IS server, so all workers subscribe to it and gets updated by IS whenever the subscription list is changed, either by a subscriber or by a worker. Existing backup functionality of IS is also utilised. In addition, all subscription data is available for monitoring by standard TDAQ tools, and the same IS server can be used to publish any other MTS specific data for monitoring. Since the load on this IS server is very low, one of existing infrastructure IS servers can be utilised, or a dedicated server added.

In case of IS server is used for sharing, the following interfaces are used to make MTS subscription:

- Worker subscribes to IS server (using confusingly named “*subscribe*” interface of IS) on the changes in MTS subscriptions data and gets notified by IS when a receiver is added or removed
- Receiver which needs to (un)subscribe to MTS messages updates data in IS, adding or removing new MTS subscription
- Worker updates IS if it needs to remove a (dead) subscriber³

4.3 MTS interfaces and collaboration with other packages

The following Illustration 1 shows a component diagram of IPC, IS and MTS packages and use of corresponding public interfaces, in order to accomplish functionality of MTS. The architecture where IS server is used for sharing MTS subscriptions is shown.

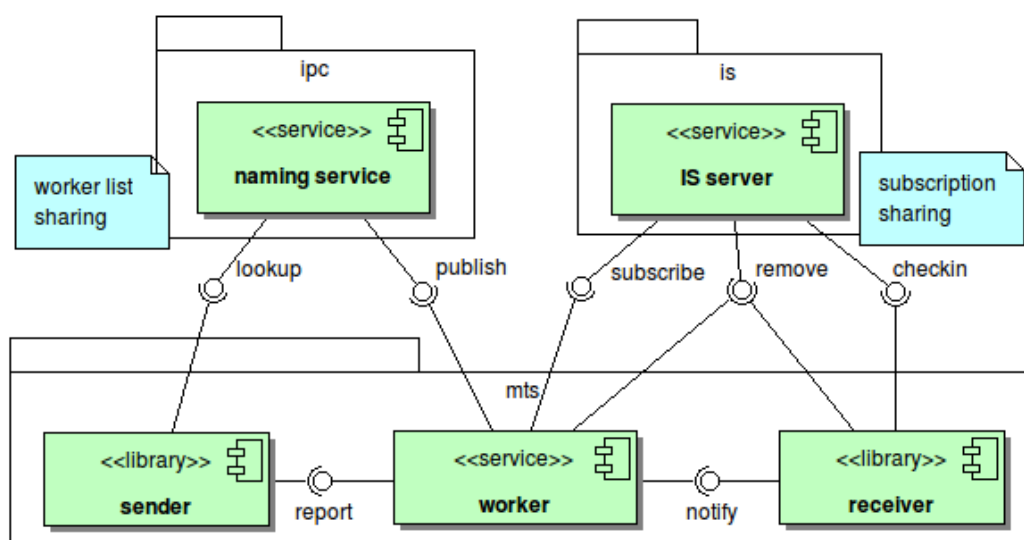


Illustration 1: IS, IPC and MTS components and interfaces

5 Detailed design

5.1 Data structures

Issue: fully corresponds to ERS Issue class:

³This can also be done externally by an operator or by e.g. expert system if it decides that a subscriber has to be removed or killed

```
struct Issue
{
    string      ID ;
    string      message ;
    unsigned long long time ;
    Severity    sev ;
    RemoteContext cont ;
    Parameters   params ;
    Qualifiers   quals ;
    CauseIssue   cause ; }

```

Subscription: structure which defines a single subscription from a receiver and stored in IS:

```
struct Subscription
{
    string      Receiver ; // CORBA reference in text format
    string      criteria ; // message selection criteria
}

```

Unique ID of a subscription is generated on receiver side (e.g. application name + PID + unique index) and used as name of IS information object to access it in IS repository (for further operations like removal).

5.2 IDL interfaces

Since the complexity of registration and subscription management is done with standard TDAQ tools IPC and IS, MTS interfaces look very simple:

```
interface worker: ipc::servant {
    void    report(in Issue an_issue); }

```

```
interface receiver {
    oneway    void    notify(in Issue an_issue); }

```

Worker inherits from `ipc::servant` in order to be accessible in IPC.

5.3 Subscription syntax

Selection criteria syntax allows to filter messages specifying selection expressions using ERS Issue attributes: severity (sev), applicationID (app), messageID (msg), qualifiers (qual), grouping of expressions and arbitrary logical combination (and, or, not) of them. Syntax is defined using EBNF notation [5]:

```
key = 'app' | 'msg' | qual
sev = 'sev'
sevlev = 'fatal' | 'error' | 'warning' | 'info' | 'information'
token = +[a-zA-Z_:\]-'*'
item = (key ('=' | '!=') token) | (sev ('=' | '!=') sevlev)
factor ken= item | ( expression ) | ('not' factor)
expression = '*' | factor * (('and' expression) | ('or' expression))

```

String comparison operations (line 4 matching key with token) supports simple pattern matching.

An example of a selection criteria is

```
(sev=ERROR or sev=FATAL) or (app=Tile* and not qual=debug)

```

5.4 MTS package Component view

MTS package is composed of:

- Sender library, implementing `ers::OutputStream` interface. This library realizes a named ('mts') ERS stream, which may be configured as output stream for a sender using ERS environment and loaded as plug-in in runtime. This part of code is completely hidden from users, which use ERS API for reporting ERS Issues.
- Receiver library, implementing `ers::InputStream` interface. Constructing ERS input stream for 'mts' transport, user must provide MTS selection criteria.
- Worker application, running on a dedicated infrastructure node. Application uses Worker class from the MTS package library.

Both receiver and sender libraries are provided in java and c++ languages.

5.5 Collaboration Scenarios

The collaboration diagram on Illustration 2 presents the main collaboration scenarios in MTS system.

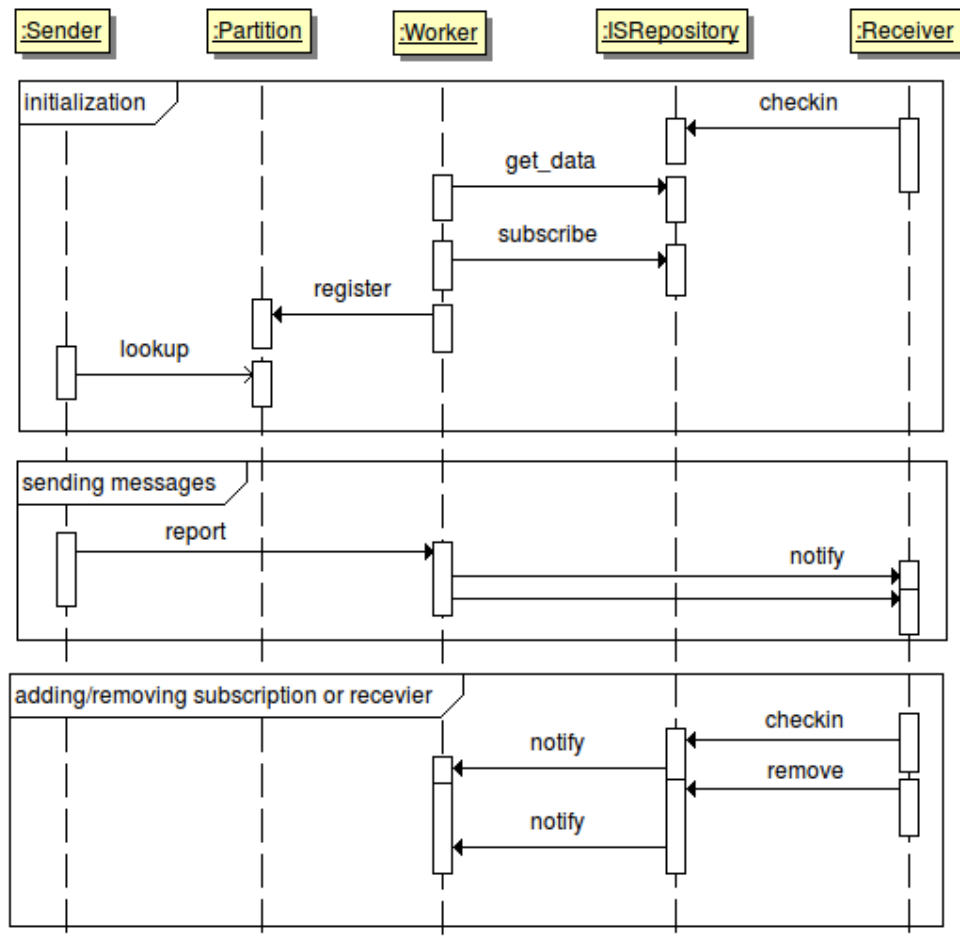


Illustration 2: Sequence diagram of MTS components activity

1. It is assumed that ISRepository (IS) and Partition (IPC) servers are launched before, as part of global partition infrastructure, taking into account necessary dependencies

2. Initialization (start or restart) of workers and receivers, adding or removing subscriptions may be done in any order, asynchronously. IS keeps all actual subscription data in central place (repository) and guarantees notification of workers in case of updates in subscription data (adding/removing subscriptions) made by receivers or other workers.
3. Workers publish its names in IPC after initialization and getting actual subscription data from IS Repository

5.6 Sender

Local queue

For simplification of the design and implementation, it is suggested not to implement buffering and detached threading on the sender side. It is assumed that sending operation is fast and never blocked at the worker side (because the buffering is done there). It may result (in rare cases of unavailability of a single worker) in a short timeout in the sender call, which is configurable and can be set as small as 1-2 seconds, after that another worker is taken.

Worker selection

Sender may use one or several (for example in case of high incoming rate) of available workers, and periodically refresh list of workers in IPC because new workers could be added dynamically, or later then the sender was started. This refresh time may be at the order of dropping idle TCP connection (few minutes), i.e. selection of new workers shall not cause unnecessary connection instantiation which is an expensive operation, neither it shall lead to a presence of too many open connections to many workers from all senders.

As a response to two-way “report” method, a worker may return to the sender some metrics value, indicating current load on that worker (in terms of queues sizes, rates or/and total system load or combination of that), in this case sender may decide to refresh and select another, less loaded worker, or to use more loaded worker less frequently w.r.t. other used workers.

5.7 Worker

5.7.1 Architecture

The main building blocks of a worker:

- pool of CORBA input threads, accepting incoming 'report' calls from senders
- output queues of messages, one queue per subscriber
- pool of worker notification threads, calling 'notify' on receivers

Input threads match each incoming message to each subscriber in the list and if a message is to be delivered to a subscriber, it is added to that subscriber's queue (using smart pointers to avoid extra copying). These threads are not IO-bound and do not block on network operations, but they perform the most CPU-intensive worker operations: subscription syntax parsing and matching. So it is reasonable to have size of this pool to be of a number of available CPU cores on the worker node.

Pool of notification threads handle output queues in a fair, e.g. in simple round-robin manner, such that each subscriber gets minimal (or equal) quantum of all available worker resources regardless of the behaviour of the other subscribers. More complex priority schema is easy to implement, for instance to count the time spent by all worker threads on each receiver or simply to judge on the queue size, decreasing the receiver priority accordingly, such that worker threads are allocated to this receiver less frequently.

5.7.2 Slow subscriber and message dropping policy

The size of output queues is limited (configuration parameter of a worker), the queue length may grow in case of some peaks in incoming rates. However in case when a subscriber is not able to accept messages at the reporting rate for some relatively long (w.r.t. incoming peaks) period of time (i.e. it is considered as 'slow subscriber'), queue may become full and old messages from the head of the queue are dropped. ~~Receiver is notified about such condition by adding a special qualifier to the following ERS issues.~~ In this case MTS Worker sends a dedicated ERS issue so that the problem may be handled by the Run Control or Online Recovery components.

In more details, the following scenarios are considered:

- *CORBA::Transient* exception is raised by remote notify method. This means the connection is destroyed and there is no application accepting messages on the receiver side. The subscriber is removed from the list and this is shared to other workers (which themselves could detect the same condition meanwhile).
- A notification thread calling one-way notify method may be blocked for a configurable period of time, then *CORBA::Timeout* exception is raised. In case of one-way notification method, it means that remote side of TCP connection is not able to accept more data, because of slowness of the application or because of the global system issues on the node. In this case the subscriber is not removed, worker thread task exits and message is kept in the queue and will be handled by the next worker thread when it is allocated to this queue.
- There are no timeouts, but receiver queue is growing such that it reaches the configured size (soft-limit of “N” messages). In this case, all the older messages exceeding N are removed from the queue, and the rest of the queue is handled.
- An input thread detects that hard limit of “M”⁴ messages in queue is reached, so it can not add more messages to it (and the workers threads are not yet there). In this case input thread removes all old messages up to “N” and places the fresh message in queue.

5.8 Receiver

Relatively simple component, which task is to restore original ERS issue from MTS message, using either `ers::IssueFactory`, or in case it is not available (i.e. this issue is not loaded), using `AnyIssue` class. Then user-supplied `ers::Receiver::receive(const Issue&)` method is called.

Receiver library implements “*notify*” method and uses CORBA-provided threading facilities and no local message queues, i.e. user callback is called directly from CORBA threads.

6 Class view

The UML class diagram on Illustration 3 includes main classes and selected attributes and relationships of MTS library and few related classes of ERS, IPC and IS packages.

4 Calculation of N and M parameters: assuming that a worker is given a memory enough to hold in total L_{total} messages (e.g. for 1Gb of memory $L_{total} = 5,000,000$), M for each receiver can be calculated dynamically: $L_{total}/2$ for the first slow receiver, for the next receiver limit is $L_{total}/4$ and so on if more slow receivers appear (a control of that total number of currently buffered messages does not exceed L_{total} is also needed). More complex algorithms like giving all buffer to the first slow receiver and then shrinking its buffer as other slow receivers come may be implemented. N can be selected as $0.9 \cdot M$.

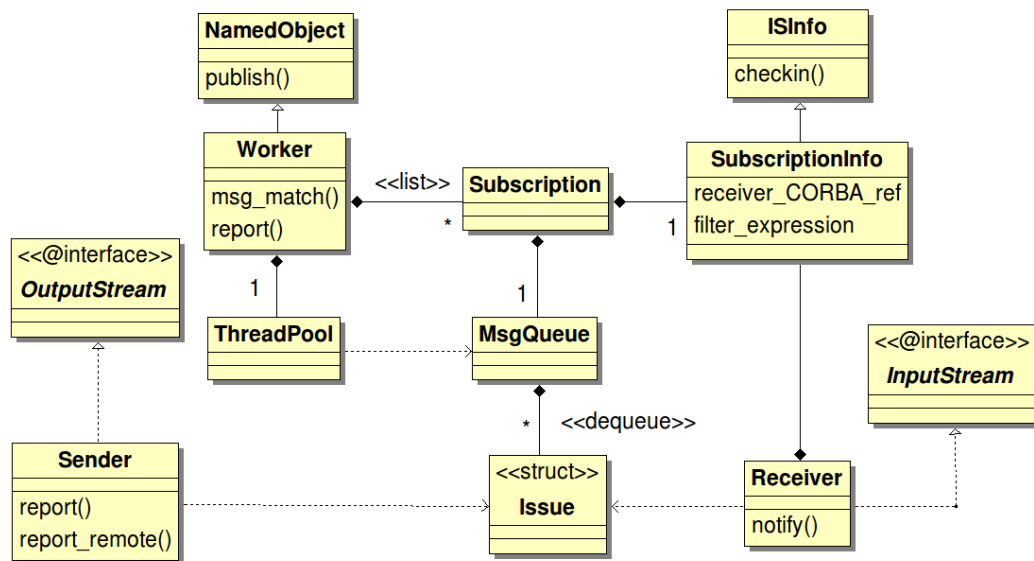


Illustration 3: classes of MTS package and selected attributes and relationships