ARMY RESEARCH LABORATORY


# Yap: A High-Performance Cursor on Target Message Router

by Jesse Kovach

ARL-TR-7096 September 2014

## NOTICES

### Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

**Army Research Laboratory**
Adelphi, MD 20783-1138

**ARL-TR-7096** **September 2014**

# Yap: A High-Performance Cursor on Target Message Router

**Jesse Kovach**
**Computational and Information Sciences Directorate, ARL**

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED (From - To) |
|---|---|---|
| September 2014 | Final | 5/2013–8/2014 |

**4. TITLE AND SUBTITLE**
Yap: A High-Performance Cursor on Target Message Router

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Jesse Kovach

**5d. PROJECT NUMBER**
R.0010376.11

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
U.S. Army Research Laboratory
ATTN: RDRL-CII-B
2800 Powder Mill Road
Adelphi, MD 20783-1138

**8. PERFORMING ORGANIZATION REPORT NUMBER**
ARL-TR-7096

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSOR/MONITOR'S ACRONYM(S)**

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
Cursor on Target (CoT) is an extensible markup language (XML) message format and associated communications protocol that has been adopted as a de facto interoperability standard for US and coalition command and control systems. CoT messages are exchanged between systems using a message router. This report describes the CoT message router implementation developed by the Battlefield Information Processing Branch of the US Army Research Laboratory.

**15. SUBJECT TERMS**
Cursor on Target, CoT, message router, CoT server, unattended ground sensors

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Jesse Kovach |
| Unclassified | Unclassified | Unclassified | UU | 26 | 19b. TELEPHONE NUMBER (*Include area code*) 301-394-3988 |

## Contents

## List of Figures

# 1. Introduction

Cursor on Target (CoT)[1] is an extensible markup language (XML) message format and associated (minimalist) communications protocol originally designed by the MITRE Corporation for the Air Force. CoT is designed to be lightweight and easy to implement, and has been adopted as a de facto interoperability standard for US and coalition command and control systems. CoT messages are generally exchanged between systems through a CoT message router, which functions as a reflector/repeater to relay messages from one system to another. A number of CoT message router implementations exist. This report describes one such message router—the Yet Another Pubsrv (Yap) application developed by the Battlefield Information Processing Branch of the US Army Research Laboratory (ARL).

# 2. Motivation

Cursor on Target messages are exchanged between systems using either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). Systems that use UDP send one CoT message per UDP datagram, with no additional headers, framing, or segmentation beyond what is included in UDP itself. Messages can be sent either point-to-point or multicast depending on the application. This approach limits the CoT message size to 64 KB (the maximum size of a UDP payload) and does not guarantee delivery. This approach works well for small, repeating messages such as Blue Force tracking position reports, but is not well suited for large messages or messages that are only sent once.

Systems that use TCP to send CoT send one message per TCP connection, again with no additional headers or framing. (This protocol is called "Open-Squirt-Close" in the CoT developer documentation.) TCP connections are always point-to-point. Some systems can send multiple messages over the same TCP connection to cut down on connection-establishment overhead, but these techniques are ad hoc and not standardized.

Two CoT systems can communicate directly by simply sending TCP and UDP messages to each other. If more than two systems need to communicate, either multicast or a CoT message router is required. This message router, often referred to as a "CoT server," serves as a reflector, receiving CoT message traffic and relaying it to a configured list of destinations.

MITRE provides a message router called Pubsrv as part of the standard CoT tools distribution. Pubsrv works well with UDP CoT traffic, but it has several limitations that make it poorly suited for use with TCP. For example, Pubsrv appears to be single-threaded and only attempts one TCP connection at a time. If the remote TCP endpoint cannot be reached, Pubsrv will hang until the

TCP connection attempt times out and will not forward any CoT traffic to any destination—TCP or UDP—during this timeout period. If the remote TCP endpoint is slow to accept connections or messages, traffic to other destinations will be delayed as well. Also, Pubsrv is a graphical user interface (GUI) Windows application, making it poorly suited for headless, unattended, or embedded applications. To work around these issues, vendors often develop their own CoT message router implementations, but few if any of these alternative implementations are generally available.[2]

For a demonstration in 2013, the Battlefield Information Processing Branch of ARL had a requirement to use CoT to exchange unattended ground sensor (UGS) observation data, including detection reports and images, with a number of other systems. The nature of this traffic required the use of TCP to accommodate large messages and guarantee delivery. We attempted to use Pubsrv to route this traffic, but encountered difficulty. We ultimately developed a custom in-house message router, known as Yap, for this demonstration by leveraging a number of CoT components and modules built for previous projects. Yap was specifically designed to perform well with TCP, but is also useful as a general-purpose CoT message router. The remainder of this report describes the design, implementation, and usage of this message router.

## 3. Requirements

Yap was originally rapid-prototyped in support of a demonstration, so no formal requirements gathering or design process was conducted. However, the system was built (and later extended as needed) to satisfy the following basic requirements:

- The message router should work well with both TCP and UDP. Specifically, problems and slowdowns with one TCP destination should not cause the message router to fail to send traffic to other destinations in a timely manner. (Practically speaking, this means the message router must use a multithreaded design.)
- The message router should support UDP multicast as well as unicast.
- The message router should support some type of filtering to control which messages are sent to different endpoints. (In some configurations, this feature is required to avoid message loops.)
- The message router should run without a GUI to allow for use in unattended or embedded applications.
- The message router should support XML validation, error checking, and traffic logging to aid in debugging CoT-enabled systems.

## 4. Features

In addition to the above, Yap contains the following features:

- Yap is written in C# and targets the .NET Framework 3.5 Client Profile. It can be run on Windows as well as Linux platforms using Mono. (It has been tested on Debian 7.)
- On Windows, Yap can either run as a console application or as a service. The same executable is used for both modes. Yap can install itself as a service (no external tools or service wrappers are required).
- The message router configuration (including global options, validation options, logging options, input and output channels, and filters) is contained in a single XML configuration file. The syntax of the configuration file is kept as simple as possible.
- A command line interface (CLI) is available that can be used to view debugging messages, control logging, enable/disable input and output channels, and reload the configuration file. The CLI is accessible through the console (if running as a console application) or optionally via telnet (over the loopback interface only).
- In addition to standard UDP and TCP, a reverse-TCP mode is available. This mode allows for remote clients to establish an inbound connection to the server and wait for messages. Multiple messages are sent over each connection, delimited by nulls. This mode is useful for working around issues with firewalls and network address translation (NAT).

Installation and usage instructions for Yap are described in the user's guide, included with this report as an Appendix.

## 5. Implementation

The Yap message router is built on top of a CoT processing library, known as CoT.Support, which ARL developed for previous efforts dating back to the 2009 Empire Challenge demonstration. The CoT.Support library contains functions for sending and receiving CoT messages but does not handle message routing. Additional modules were developed to provide message routing functionality. This section discusses the existing and new components used in the Yap implementation and describes the data flow of messages within the application.

### 5.1 Input

Message input to Yap is handled by one or more *InboundChannels*. The *InboundChannel* class encapsulates a *CotLowLevelListener* along with configuration information. The

*CotLowLevelListener* class is a generic CoT receiver class from the CoT.Support library. It encapsulates two threads that receive TCP and UDP messages (one protocol per thread) using the standard .NET *TcpListener* and *UdpClient* classes. Received messages are handed off to the main program via a C# event for further processing, as shown in Fig. 1.

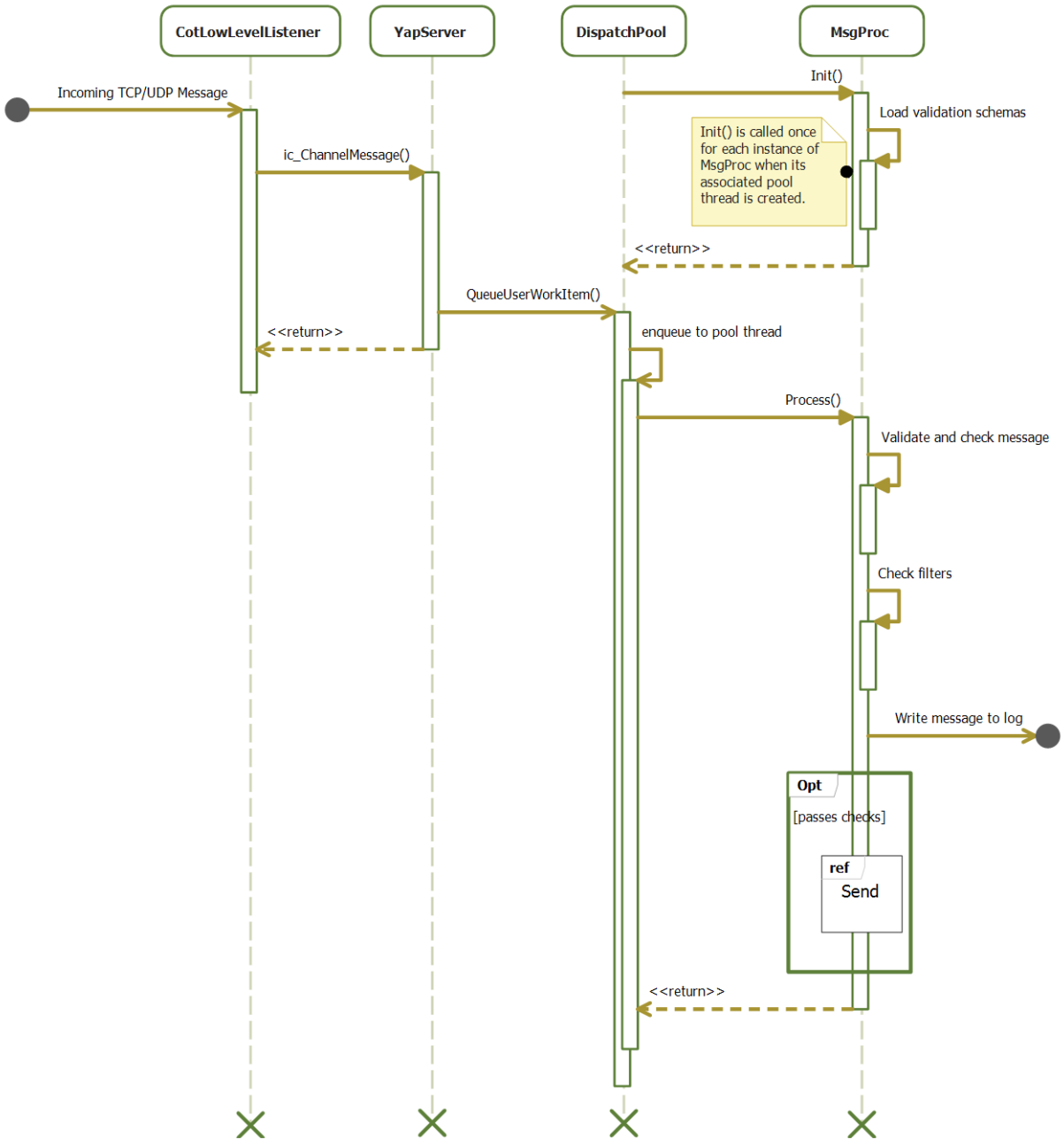


Fig. 1 Input message data flow

## 5.2 Processing and Filters

Message processing within Yap is handled by the *MsgProc* class in conjunction with the *DispatchPool* custom thread pool class. The *DispatchPool* creates a number of threads (by

default equal to the number of processors in the system) and creates an instance of *MsgProc* for each thread.[3] The Yap main program calls DispatchPool.QueueUserWorkItem() to place received CoT messages in a queue for processing by the *MsgProc* threads. The *DispatchPool* removes messages from this queue and calls the MsgProc.Process() method to process each message as threads become available.

*MsgProc* performs a number of validation checks on incoming messages. *MsgProc* first validates the incoming message against the CoT XML schemas and then attempts to parse the messages. Validation and parsing errors are logged, and messages that fail validation can optionally be dropped based on configuration settings. (The validation and parsing can also be disabled entirely to improve throughput if needed, but this will disable all message sanity checks as well as filters that rely on checking the fields of the parsed message.)

After the message is validated and parsed, a number of sanity checks are performed on the time fields in the CoT message to attempt to detect clock skew between systems or other configuration/coding errors. *MsgProc* checks for timestamps in the future or far past and for stale times in the recent past. Warnings are logged for any message that fails these checks, but the messages are still processed. The thresholds for these sanity checks are configurable.

After the sanity checks are complete, the message is checked against the input filters (if any) configured for the input channel that generated the message. (See the user manual in the Appendix for a detailed discussion of the available filters.) Messages that do not match the filters are dropped. Once the filter check is complete, the message is written to the CoT log if the log is enabled (dropped messages are logged as well).

Finally, any message that was not marked as dropped is sent to the outbound channels as described below.

### 5.3 Output

Message output in Yap is handled by one or more *OutboundChannels*. The *OutboundChannel* class, similar to the *InboundChannel* class, encapsulates a *CotSender* as well as configuration information. The *CotSender* class was part of the preexisting CoT.Support library and was extended to support additional features for Yap.

The MsgProc.Process() method passes messages to outbound channels by calling OutboundChannel.Send() on each channel, as shown in Fig. 2. The *OutboundChannel* then checks the message against the output filter list (if any) configured for that channel. Messages that do not match the filter list are dropped. Otherwise, the *OutboundChannel* calls the Send() method on the underlying *CotSender* to enqueue the message for transmission.

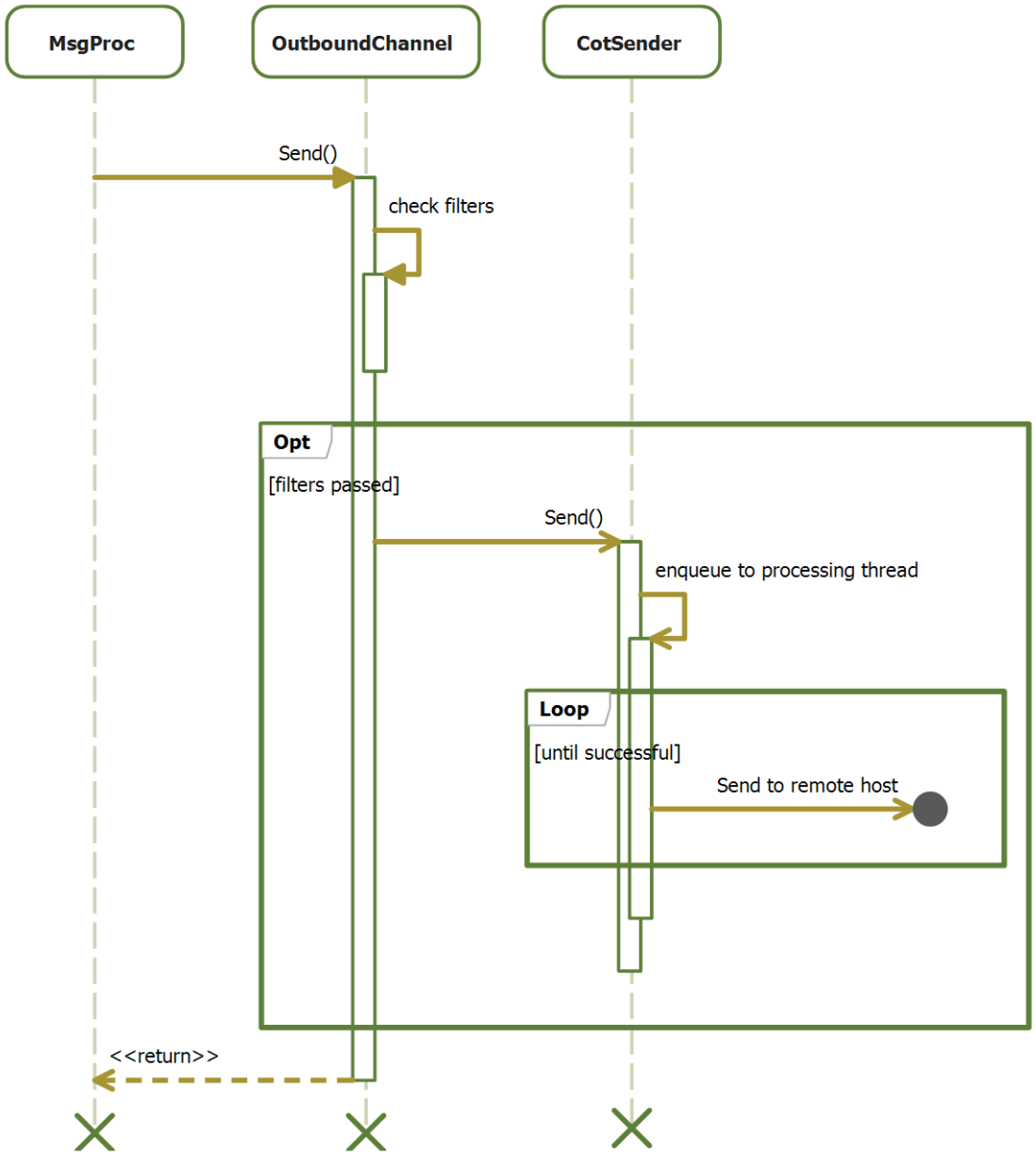


Fig. 2 Output message data flow

The *CotSender* class encapsulates a processing thread that reads messages from that sender's output queue and attempts to send them to the configured destination or multicast group. For UDP messages, there is no error detection or recovery (in line with UDP semantics). However, there is a configurable inter-message delay between UDP send attempts to work around issues with systems or networks that drop messages if too many are sent together. For TCP messages, if a send attempt fails, the message is either requeued or dropped based on configuration settings.

## 6. Conclusions and Future Work

The Yap CoT server is a high-performance CoT message router designed to work effectively with TCP as well as UDP.

Further work in this area could involve conducting performance testing of Yap with the aim of quantifying the performance versus other CoT server implementations, as well as fixing any performance issues identified during testing. Also, Yap could potentially be extended to support an application programming interface (API) for loading filters and channel modules at runtime, enabling the server to be further customized for specific applications without needing to modify and recompile the entire system.

## 7. References and Notes


1. Butler M. The Developer's Guide to Cursor on Target. The MITRE Corporation: Bedford, MA, August 2005.
2. I am aware of alternative CoT message router implementations from WinTec Arrowmaker and several different groups within MITRE. Other implementations may exist as well.
3. The .NET runtime provides a thread pool class as part of the standard library. Yap does not use the standard thread pool, because it needs to maintain local state for each thread and the standard thread pool class does not allow for this. Specifically, in order to perform XML validation, each thread needs a local instance of an *XmlSchemaSet* class containing the validation schemas as the documentation states that this class is not thread safe. (An alternative approach would be to create a new *XmlSchemaSet* for each message. This would allow for the use of the standard thread pool, but would affect performance as the rather extensive collection of CoT XML schemas would have to be reloaded for every message.)

# Appendix. Yap User's Guide

## A-1 Installation

For maximum flexibility, Yet Another Pubsrv (Yap) is distributed as a zip file containing an executable and supporting dynamic-link libraries (DLLs) and does not include an installer. Simply unzip the file to a location of your choice. Yap requires the .NET Framework 3.5 Client Profile on Windows or a compatible version of Mono on Linux. It has been tested with Mono 2.10 as distributed by Debian 7, but other versions should work as well if they are compatible with .NET 3.5.

## A-2 Configuration File

Yap is configured using a configuration file. An example configuration file named yap.conf.example is included with the distribution. This file should be copied to yap.conf and edited as needed to match the desired configuration. See Section A-6 for details.

## A-3 Installation as Windows Service

Yap may optionally be installed as a Windows service. To install as a service, open an administrator command prompt and run the following:

```
yap –i [name of configuration file to use]
```

If the configuration file name is not specified, then it will default to loading yap.conf from the current directory.

Example:

```
C:\yap>yap -i yap.conf
Trying to install Yap service: C:\yap\Yap.exe -s C:\yap\yap.conf
Service installed and started
```

If you receive a "Could not connect to service control manager" error, make sure that you have administrator rights.

If you receive an "Unable to start service" error, make sure that the configuration file exists and is valid. The service will not start if the configuration file contains errors or is missing.

## A-4 Removal of Windows Service

To remove the Yap service, open an administrator command prompt and run the following:

```
yap -u
```

Example:

```
C:\yap>yap -u
Trying to uninstall Yap service
Service uninstalled
```

### A-5 Running as a Console Application

If Yap was not installed as a service, it can be run as a standard Windows application as follows:

```
yap [name of configuration file]
```

If no configuration file is specified, Yap will load the file “yap.conf” from the current directory.

Example:

```
C:\yap>yap yap.conf
[21 Jul 17:21:13] Loading configuration from C:\yap\yap.conf
[21 Jul 17:21:14] Configuration reloaded
[21 Jul 17:21:14] Yap 0.6.0.0 is running
yap>
```

If the configuration file is missing or contains errors, an error message will be displayed and Yap will exit. Correct the configuration file and try again.

### A-6 Configuration

The Yap configuration is specified in an extensible markup language (XML) file. The file defines inbound and outbound channels, as well as global server options and logging options. Inbound channels process incoming Cursor on Target (CoT) data from other systems. Outbound channels send CoT messages to other systems. Filters can be set on both inbound and outbound channels. At least one inbound channel and one outbound channel must be defined in order for Yap to pass message traffic.

A sample configuration is shown below. This example configuration file is also provided with the Yap distribution.

```
<?xml version="1.0" encoding="utf-16"?>
<yapConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <global
      remoteCmdEnabled="true"
      remoteCmdPort="17999"
      futureCheckSecs="15"
      pastCheckSecs="15"
      staleCheckSecs ="86400"
      validationEnabled="true"
      validationFailOnWarning="false">

    <validationSchema>public-schemas/cot/Event.xsd</validationSchema>
    <validationSchema>public-schemas/cot/Event.xsd</validationSchema>
    <validationSchema>public-schemas/cot/CoT__flow-tags_.xsd</validationSchema>
    <validationSchema>public-schemas/cot/CoT_contact.xsd</validationSchema>
    <validationSchema>public-schemas/cot/CoT_image.xsd</validationSchema>
```

```
  <validationSchema>public-schemas/cot/CoT_link.xsd</validationSchema>
  <validationSchema>public-schemas/cot/CoT_remarks.xsd</validationSchema>
  <validationSchema>public-schemas/cot/CoT_request.xsd</validationSchema>
  <validationSchema>public-schemas/cot/CoT_sensor.xsd</validationSchema>
  <validationSchema>public-schemas/cot/CoT_shape.xsd</validationSchema>
  <validationSchema>public-schemas/cot/CoT_spatial.xsd</validationSchema>
  <validationSchema>public-schemas/cot/CoT_track.xsd</validationSchema>
  <validationSchema>public-schemas/cot/CoT_uid.xsd</validationSchema>
</global>

<logging
  cotLogEnabled="false"
  cotLogDir="LogFiles/Yap"
  cotLogBaseName="CoT"
  progLogEnabled="false"
  progLogFileName="yap.log "/>

<inboundChannels>
  <!-- example without filters -->
  <inboundChannel name="In0" port="18000" />

  <!-- example with filters -->
  <inboundChannel name="In1" port="18001" listenIP="0.0.0.0" tcp="true"
   udp="true" parse="true" validate="true" rejectIfValFail="true" >
    <filters matchMode="matchAny">
      <filter xsi:type="ipFilter" ip="172.18.128.0/24" />
    </filters>
  </inboundChannel>

  <!-- multicast example -->
  <inboundChannel name="In2" port="18002" multicastIP="239.2.2.1" tcp="false"
   udp="true" />
</inboundChannels>

<outboundChannels>
  <!-- example without filters -->
  <outboundChannel name="Out0" host="172.18.134.10" portUdp="18010"
   portTcp="18010" protocol="udp"/>

  <!-- example with filters -->
  <outboundChannel name="Out1" enabled="true" greedy="false"
   host="172.18.30.10" portUdp="18010" portTcp="18010"
   protocol="Udp" udpSizeLimit="32000" udpDelayMsec="50"
   maxQueueLength="50"
   sendFailBehavior="requeueFront" queueFullBehavior="dropNewest" >
    <filters matchMode="matchAny" onError="drop">
      <!-- nested filter list -->
      <filter xsi:type="filterList" matchMode="matchAll" >
        <filter xsi:type="geoFilter" nwLat="0.2" nwLon="0.1"
         seLat="0.1" seLon="0.2" />
        <filter xsi:type="regexFilter" field="opex" regex=".*" />
      </filter>
      <filter xsi:type="ipFilter" ip="172.18.30.0/24" />
      <filter xsi:type="channelFilter" channel="In2" />
    </filters>
  </outboundChannel>

  <!-- multicast example -->
  <outboundChannel name="Out2" host="239.2.2.1" portUdp="18002"
   protocol="udpOnly">
    <filters matchMode="matchNone">
      <!-- needed to avoid message loops -->
      <filter xsi:type="channelFilter" channel="In2" />
```

```
        </filters>
      </outboundChannel>

      <outboundChannel name="Out3" portTcp="18003" protocol="tcpListen"/>
    </outboundChannels>

</yapConfig>
```

Configuration options are described in detail in the following sections.

## A-7 Global Settings

These settings adjust various parameters and apply to all channels on the server.

`remoteCmdEnabled` – enables or disables the telnet command line interface (CLI). This is useful when Yap is running as a server as it allows access to the CLI to reload the configuration or check channel status. Note that there is **no authentication** on the CLI, but it is only accessible via the loopback interface.

`remoteCmdPort` – sets the port used by the telnet CLI.

`futureCheckSecs, pastCheckSecs, staleCheckSecs` – if logging is enabled, these parameters are used to check for clock skew as follows:

- If the "time" element of a message is more than `futureCheckSecs` seconds in the future, a warning will be logged.
- If the "time" element of a message is more than `pastCheckSecs` seconds in the past, a warning will be logged.
- If the "stale" element is **more** than `pastCheckSecs` seconds in the past but **less** than `staleCheckSecs` seconds in the past, a warning will be logged. This avoids nuisance warnings on intentional deletes that have stale times in the far past by design.

`validationEnabled` – enables schema validation of CoT messages on inbound channels. This can be set to true or false. If true, validation will be enabled globally (validation must also be enabled in the configuration for an inbound channel to validate messages for that channel). If false, validation will be disabled globally for all channels. The built-in .NET XML libraries are used for validation. Enabling validation is useful for debugging and troubleshooting, but **will significantly reduce the throughput of the server.**

`validationFailOnWarning` – controls whether a CoT message that triggers a validation warning will be treated as having failed validation. This can be set to true or false. Validation warnings are generated when a schema cannot be found to validate a particular element but the XML is otherwise well formed. (For example, this may occur if a custom subschema is being used.)

`validationSchema` – specifies a schema to be used for validation. This element can occur multiple times in the configuration file. The list of validation schemas included with the default

configuration contains the main CoT event schema as well as all public subschemas. Additional subschema definitions can be added to enable validation against those subschemas. At least one schema must be provided in order for validation to occur.

### A-8 Logging Settings

These settings control the logging of CoT message traffic and program error and debug messages. These settings apply to all channels on the server.

`cotLogEnabled` – enables or disables logging of CoT message traffic.

`cotLogDir` – sets the directory where CoT message logs will be written. Log file names are generated automatically in this directory based on the date. A new log file is created daily.

`cotLogBaseName` – sets the initial portion of the generated log file names. For example, a value of "CoT" will result in filenames such as `CoT-2010.12.31.log`

`progLogEnabled` – enables or disables program debug/error logging.

`progLogFileName` – sets the filename used for the debug/error log.

### A-9 Inbound Channels

Inbound channels listen for messages using Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). Messages received on inbound channels are sent to all outbound channels once they pass any filters that may be configured on the inbound channel. Any number of inbound channels can be configured.

`name` – display name for the channel. This parameter is required.

`port` – TCP/UDP listen port for the channel. This parameter is required.

`listenIP` – specifies the local listen Internet Protocol (IP) address to use for the channel. This parameter is optional. If not specified, the channel will listen on all interfaces.

`multicastIP` – specifies a multicast group to join. This parameter is optional. If present, multicast CoT traffic from this group will be received by this channel. If not specified, multicast will not be used for this channel.

`tcp`, `udp` – set to true to enable the corresponding protocol or false to disable. These parameters are optional and will default to true if not present. Both TCP and UDP can be enabled simultaneously for a unicast channel (the same port will be used for both). TCP is not supported for multicast channels.

`parse` – if set to true, incoming CoT messages on this channel will be parsed by an XML parser. This will ensure that the XML is well formed and is also necessary for certain filters to function and for validation to occur, but will also slow down throughput. If set to false, messages will not be parsed. This parameter is optional and will default to true.

`validate` – if set to true, incoming CoT messages on this channel will be validated against the CoT schemas provided that validation is enabled and configured in the global settings. If set to false, messages on this channel will not be validated. Parsing must be enabled for validation to occur. This parameter is optional and will default to true if not specified. Note that schema validation is resource-intensive and will significantly slow throughput.

`rejectIfValFail` – controls the handling of messages that fail validation or parsing. If set to true, messages that do not parse or validate will be dropped. If set to false, messages that do not parse or validate will still be processed and sent to outbound channels (but a warning will be logged if logging is enabled).

`filters` – specifies an optional filter list to apply to this channel. If a filter list is specified, messages that arrive on this channel and do not match the filter will be dropped. See the Filters section below for details.

### A-10 Outbound Channels

Outbound channels send messages to remote systems. Filters can be specified on each channel to control the flow of data to each system. Any number of outbound channels can be configured.

`name` – display name for the channel. This parameter is required.

`host` – the IP address or hostname of the remote endpoint or the address of the multicast group to which CoT messages will be sent. This parameter is required for all protocol options except tcpListen. If a multicast address is specified the protocol must be `udpOnly`.

> **CAUTION:** If the server is configured with an inbound and outbound channel on the same multicast group, a loop will be created and a message storm may result. A filter can be specified on the outbound channel to avoid this problem. See the In2 and Out2 channels in the example configuration above.

`protocol` – the protocol to use for the channel. This parameter is required. Choices are as follows:

- `tcp` – uses TCP with one message per connection ("Open-Squirt-Close" as described in the CoT developers guide).
- `udp` – uses UDP with TCP fallback. If a message exceeds `udpSizeLimit`, it will be sent using TCP.
- `udpOnly` – uses UDP only. If a message exceeds `udpSizeLimit`, it will be dropped.
- `tcpListen` – uses TCP in listen mode. The channel will listen for inbound TCP connections on the specified TCP port. Clients can connect to this port to receive messages. Messages will be sent to all connected clients. When using this mode, multiple messages will be sent over the same connection and messages will be delimited using a zero byte

after each message. This is a nonstandard extension but is useful when building systems that need to connect through firewalls and network address translation (NAT) routers.

`portTcp` – specifies the port to use for TCP. This parameter is required if the protocol is `tcp`, `tcpListen`, or `udp`.

`portUdp` – specifies the port to use for UDP. This parameter is required if the protocol is `udp` or `udpOnly`.

`enabled` – true to enable sending messages on this channel, false to disable the channel. This parameter is optional and defaults to true if not present. Channels can also be enabled and disabled at runtime using the CLI.

`greedy` – if set to true, messages that are sent out this channel (i.e., the channel is enabled and the message passes any filters set on the channel) will not be processed further and will not be sent out any subsequent channels. This is useful in conjunction with filters. Channels are evaluated in the order in which they appear in the configuration file. This parameter is optional and defaults to false if not present.

`udpSizeLimit` – specifies the maximum message size that will be sent using UDP. This parameter is optional and defaults to 32000 if not specified.

`udpDelayMsec` – specifies a delay in milliseconds between UDP messages to work around issues with systems or networks that drop packets if they are sent too quickly. This parameter is optional and defaults to 50. It can be set to 0 to disable the delay.

`maxQueueLength`, `sendFailBehavior`, `queueFullBehavior`– these options control the behavior of the send queue for this channel. The send queue primarily affects TCP channels, but is used for UDP channels as well due to the inter-message delay described above. The queue works as follows:

- New messages for the channel are placed at the end of the send queue.
- The maximum length of the send queue is specified by `maxQueueLength`. The default `maxQueueLength` is 300.
- If the send queue is full, the action taken is controlled by `queueFullBehavior`:
    - `dropNewest` – the new message is dropped.
    - `dropOldest` – the message at the head of the queue is dropped and the new message is inserted at the end of the queue.
    - The default option is `dropNewest`.
- Messages are sent out one at a time from the head of the queue.

- If sending a message fails (due to the remote endpoint not accepting the connection or some other error), the action taken is controlled by `sendFailBehavior`:
    - `drop` – the message is dropped.
    - `requeueFront` – the message is requeued at the head of the queue (it will be retried immediately).
    - `requeueBack` – the message is requeued at the end of the queue (it will be retried later, after all other messages in the queue are processed).
    - The default option is `drop`.

`filters` – specifies an optional filter list to apply to this channel. If a filter list is specified, this channel will only send messages that match the filter. See section A-11 for details.

### A-11 Filters

Filters can be used to control the flow of messages between channels. A filter list can be specified for inbound or outbound channels. If a filter list is specified for an inbound channel, incoming messages must match the filter list to be processed. Messages that do not match the filter list will be dropped entirely and will not be sent to any outbound channels. If a filter list is specified for an outbound channel, that channel will only send messages that match its filter list. Filter lists can be nested.

A `filterList` is composed of one or more filters and contains a required `matchMode` option and an optional `onError` option. The `matchMode` option is one of the following:

- `matchAny` – this filter list will match messages that match any one of its filters (or).
- `matchAll` – this filter list will only match messages that match all of its filters (and).
- `matchNone` – this filter list will only match messages that do not match any of its filters (negated or).

The `onError` option defaults to drop and is one of the following:

- `accept` – if an error occurs while processing the filters the message will be considered as having matched the filter list.
- `drop` – if an error occurs while processing the filters the message will be considered as having not matched the filter list.

Four filter types are available:

- `channelFilter` – matches a message that was received on a particular inbound channel (this filter is only useful on outbound channels). The `channel` parameter specifies the name of the inbound channel.

- `ipFilter` – matches a message that was received from a particular IP address or subnet. The `ip` parameter specifies either an IP address or a subnet using `192.168.0.0/16` notation. Hostnames are not supported.
- `geoFilter` – matches a message with a latitude and longitude inside a given bounding box. The `nwLat`, `nwLon`, `seLat`, and `seLon` parameters specify the four corners of the bounding box in WGS84 decimal degrees (the same notation as used in the CoT message itself). Standard math operations are used to determine whether the coordinates lie within the bounding box; no geodetic computations are done.
- `regexFilter` – matches messages where a specified field of the event element matches a regular expression. The `field` parameter specifies the field to match and can be one of `access`, `type`, `uid`, `qos`, or `opex`. The `regex` parameter specifies the regular expression to match against. The regular expression is processed using the standard .NET regular expression engine, which is compatible with Perl regular expressions.

### A-12 Command Line Interface

Yap contains a CLI, which can be used to display program status, control debug logging, reload the configuration, and enable or disable channels. The command line is accessible through telnet if this option is enabled in the configuration file. (Note that there is no authentication or encryption on the telnet CLI; however, it is only accessible from the local machine via the loopback interface.) The CLI is also accessible via the console if Yap is run as an application. If Yap is run as a Windows service, the CLI can only be accessed via telnet.

To connect to the telnet CLI interface (if enabled), telnet to 127.0.0.1 using the port specified in the configuration file.

The following are the CLI commands.

`status` – prints some statistics and traffic counts about the configured channels. This will also show information about clients connected to TCP listen channels. Example status output is shown below.

```
yap> status
Validation enabled

Inbound Channels:
In0 received=0 filtered=0 errors=0 rejects=0
  time problems=0  validation failures=0  parse errors=0
In1 received=0 filtered=0 errors=0 rejects=0
  time problems=0  validation failures=0  parse errors=0
In2 received=0 filtered=0 errors=0 rejects=0
  time problems=0  validation failures=0  parse errors=0

Outbound Channels:
Out0 enabled=True queueLen=0/300 total=0 sent=0 dropped=0
   errors=0 retries=0 filtered=0
OutListen1 enabled=True queueLen=0/300 total=0 sent=0 dropped=0
   errors=0 retries=0 filtered=0
```

```
1 clients
   127.0.0.1:1754
```

`flush` – flushes any buffered log file output to disk.

`reload` – reloads the configuration file. If the configuration file contains errors, Yap will revert to the previous configuration. (Note that if the configuration file was changed to disable the telnet CLI this setting will not be effective until Yap is restarted. All other configuration changes will take effect immediately.)

`shutdown` – exits Yap. This command is only available from the console CLI and is not available via telnet.

`quit` – exits the CLI session while leaving the server running. This command is only available via telnet.

`enable <channel-name>` – enables the channel with the specified name.

`disable <channel-name>` – disables the channel with the specified name. Messages will not be sent over the channel until it is re-enabled.

`kick <address>` – Disconnects a client from a tcpListen channel.

`debug [on|progress|off]` – Sets the debug level. Debug messages will be displayed to the CLI and will also be written to the program log if enabled in the configuration. On will enable all debug messages, progress will enable some debug messages, and off will disable all debug messages (errors and warnings will still be shown). Entering debug without a level will display the current debug level.

`validate [on|off]` – Enables or disables validation for all channels. Entering validate without an on/off setting will display the current setting.

`read <filename> [channel-name]` – Reads a CoT message from the specified file and processes it as if it was received over an inbound channel. If a channel-name is specified it will be processed using the specified inbound channel, otherwise it will be processed using the first inbound channel. Useful for testing.

## List of Symbols, Abbreviations, and Acronyms

| | |
|---|---|
| API | application programming interface |
| ARL | US Army Research Laboratory |
| CLI | command line interface |
| CoT | Cursor on Target |
| GUI | graphical user interface |
| IP | Internet Protocol |
| NAT | network address translation |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| UGS | unattended ground sensor |
| XML | extensible markup language |
| Yap | Yet Another Pubsrv |

| | |
|---|---|
| 1 (PDF) | DEFENSE TECH INFO CTR<br>ATTN DTIC OCA (PDF) |
| 2 (PDF) | US ARMY RSRCH LABORATORY<br>ATTN IMAL HRA MAIL & RECORDS MGMT<br>ATTN RDRL CIO LL TECHL LIB |
| 1 (PDF) | GOVT PRNTG OFC<br>ATTN A MALHOTRA |
| 7 (PDF) | US ARMY RSRCH LAB<br>ATTN RDRL CII B<br>J KOVACH<br>TIM GREGORY<br>ROBERT WINKLER<br>SOMIYA METU<br>RDRL SES A<br>GARY STOLOVY<br>JEFF HOUSER<br>RDRL CII C<br>MARK THOMAS<br>RDRL CII<br>BARBARA BROOME<br>DEBORAH A WELSH |
| 1 (PDF) | DEFENSE INTELLIGENCE AGENCY<br>ROBERT HEATHCOCK |
| 1 (PDF) | MITRE<br>JON JACOBY |
| 1 (PDF) | CERDEC NVESD<br>BLAKE AYCOCK |
| 1 (PDF) | UNAFFILIATED (ARL RETIREE)<br>LARRY TOKARCIK |