Cursor on Target
Technical Implementation Guide

Compliments of:
Theta Informatics LLC
CAGE: 9Q433
matthew.krupczak@theta.limited
bobby.krupczak@theta.limited
December 2023

Executive Summary

The Cursor on Target (abbreviated CoT) is a simple, powerful, and extensible network message format. The MITRE Corporation describes it as “the ‘common language’ for tactical systems that is critical in communicating much needed time sensitive position information”. This protocol is the backbone of the prolific Team Awareness Kit software released by the TAK Product Center, which allows tactical teams such as law enforcement and small unit military groups to develop situational awareness on the location of friendly and un-friendly forces alike. For the warfighter, this can have a profound effect on their ability to plan, coordinate, and effectively accomplish their given mission.

Despite the utility and adaptability of this protocol, documentation on it from official sources remains sparse and incomplete. This poses a vexing challenge for technical implementers of networked software solutions using the protocol. In Theta Informatics’ development of the *OpenAthena* software (which allows common drones to spot and share precise locations from drone images), we collected information from multiple resources and assembled basic reference implementations written in Java, Swift, and Python which are compatible with TAK and other software using the protocol.

The aim of this guide is to provide you, the reader, with the knowledge, tools, and code you need to implement this protocol effectively in your own software.

The first part of this guide consists of an overview of the CoT message format, describing its component syntax, data format, and transmission methods. The second section will consist of a walk-through of our open source Cursor on Target reference implementations written in Java, Swift, and Python using only imports from the standard libraries. The third section will contain links to and a discussion of external resources and guides for usage of the CoT protocol and a discussion on each of their utility.

Our hope is that this guide enables you to develop networked software solutions which aid the effectiveness and safety of the warfighter. At Theta, we are obsessed on the needs and requirements of this end user. More participants in this software ecosystem will accelerate technological development and improve outcomes for those serving in this critical role.

The Cursor on Target Message Format

Cursor on Target messages are written in XML and their content and format are governed by a set of XML Schemas that are registered with the DoD XML Registry.

What is XML? It is an acronym for Extensible Markup Language. Wikipedia defines XML as:

> . . . a markup language and file format for storing, transmitting, and reconstructing arbitrary data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The World Wide Web Consortium's XML 1.0 Specification of 1998 and several other related specifications—all of them free open standards—define XML.

Using XML (Extensible Markup Language) and XML Schema is an excellent choice for many data representation and exchange applications due to several compelling reasons. First, XML is highly flexible and customizable, allowing for the creation of complex data structures that can be easily understood and manipulated. Second, XML Schema, a powerful tool for defining the structure, content, and semantics of XML documents, ensures data integrity and validation. It provides a clear set of rules for what the XML document should contain, which is invaluable for data consistency, especially in systems where multiple parties are exchanging data. Furthermore, XML is platform-independent and supports Unicode, making it ideal for internationalization and ensuring compatibility across different systems and languages. Lastly, the widespread adoption of XML means that there are abundant resources and tools available for working with XML and XML Schema, including parsers, validators, and transformation tools, which simplifies development and integration processes.

Below are two Cursor on Target *event* messages. The first one was sent by *OpenAthena* Android while the second was sent by *OpenAthena* iOS after a user of each respective application analyzed a specific drone image, identified an object, and sent the object coordinates as a CoT *event* message.

The *uid* parameter specifies a globally unique identifier for this event. *OpenAthena* applications create this identifier by using the underlying platform UUID libraries along with the prefix *OpenAthena*. The type of event *a-p-G* indicates that it is an assumed-friend, is pending, and indicates an object on the ground. The *how* field is intended to give a hint as to how the coordinates were generated with *h-c* which means a human identified an object in a drone image and used *OpenAthena* to calculate its coordinates. Finally, the *time* value indicates when the event was generated (e.g. when the drone image was taken) while *start* and *stale* define an interval in time for which the event is valid.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <event version="2.0" uid="OpenAthena-49960de5-96"
                  type="a-p-G"
                  how="h-c"
                  time="2023-04-29T18:37:17Z"
                  start="2023-09-12T20:04:28Z"
                  stale="2023-09-12T20:09:28Z">
     <point lat="29.204774358887512"
           lon="-80.99900955606009"
           ce="27.439030363325003"
           hae="40.06260526340341"
           le="5.0"/>
     <detail>
       <precisionlocation altsrc="DTED2" geopointsrc="GPS"/>
       <remarks>Generated by OpenAthena for Android from sUAS data</remarks>
     </detail>
    </event>
```

Event messages must contain a p*oint* element or coordinate and optionally contain a d*etail* element*.* The *point* element must contain longitude and latitude in WGS 84 degrees, height above ellipsoid (WGS) in meters, a circular error *ce* area around the point in meters, and a linear error *le* altitude range about the point in meters. Finally, the optional d*etail* element contains information that helps communicate how the coordinate values were calculated – *OpenAthena* uses GPS metadata along with DTED2 for terrain altitude information.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <event version="2.0"
        uid="openathena-ios+B3A144B6-3A44-4C67-A222-4319FCBA60F7+1694699643"
          type="a-p-G"
          how="h-c"
          time="2022-04-13T17:52:06Z"
          start="2023-09-14T13:54:03Z"
          stale="2023-09-14T13:59:03Z" >
     <point lat="60.000124"
         lon="5.682066"
         ce="24.278165"
         hae="-471.804702"
         le="5.0" />
     <detail>
       <precisionLocation altsrc="DTED2" geopointsrc="GPS" />
       <remarks>Generated by OpenAthena for iOS from sUAS data</remarks>
     </detail>
   </event>
```

Reference Implementations: Building and Sending CoT Messages

Theta Limited's *OpenAthena* can send CoT *event* messages when its users initiate them. Because Android and iOS versions support CoT, Theta maintains code in Java (for Android) and Swift (for iOS). In addition, Theta maintains testing tools, in Python, and documents related to CoT. Thus, Theta maintains CoT related code for three different programming languages – Java, Swift, and Python – and all are publicly available on GitHub and open sourced.

By default, all versions of *OpenAthena* send CoT messages over UDP to port 6969 via IPv4 multicast group 239.2.3.1. While there is no pre-defined nor registered assignments for well-known IP addresses and ports, the TAK Product Center's reference implementation uses udp://239.2.3.1:6969 by default so that is what Theta has coded its implementations to. If you run your own TAK server, you are, of course, free to choose other ports, IP addresses, or multicast groups assuming you have permission to do so.

Because the default destination is a IPv4 multicast group, our underlying code must properly setup and configure access to IPv4 multicast socket APIs. Doing so is specific to the platform and programming language and is covered in each language/platform subsection.

Overall, each code base must:

1. setup network APIs
2. build an CoT *event* XML document
3. convert, or serialize, that XML document to a string of characters
4. send that string over the network

Java/Android Code Base

Theta's Java/Android implementation of CoT can be found in the file *CursorOnTargetSender.java* within the *OpenAthenaAndroid* GitHub repo. The main entry point to this CoT code is the method (or function) *sendCoT()*. Call this function with the latitude, longitude, altitude, theta (gimbal pitch degree), and the date/time of the original drone image and everything else is taken care of. That function first validates its arguments and then calculates a 5-minute time interval during which the CoT event is to be valid. It then calculates a globally unique UID by combining the prefix *OpenAthena* with a host-specific hash calculation plus a monotonically increasing event identifier.

```
public class CursorOnTargetSender {

  public static void sendCoT(double lat, double lon, double hae, double theta, String exif_datetime) {

      // set eventuid as long time in ms since Jan 1st, 1970
      . . . . .

      // create time interval params of 5 minutes from now
      . . . . .
```

```
        new Thread(new Runnable() {

            public void run() {

               String xmlString = buildCoT(uidString, interval params, lat, lon, ce, hae, le)

               deliverUDP(xmlString)
            }
        }).start();
    }

    public static String buildCoT(String . . . .) {

       try {

          DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance()
           DocumentBuilder db = dbf.newDocumentBuilder()
           Document doc = db.newDocument()

          // create elements, add attributes . . .
          Element root = doc.createElement("event")

          . . . . .

          // seralize to string
          TransformerFactory tf = TransformerFactory.newInstance();
          Transformer t = tf.newTransformer()

          . . . . . .

          return writer.toString()

       } catch (Exception e) {
         return "";
       }
    }

    public static deliverUDP(String xml) {

       new Thread(new Runnable() {

           public void run() {

               try {
                 MulticastSocket socket = new MulticastSocket()
                  DatagramPacket packet = new DatagramPacket(message params . . );
                  socket.send(packet)
                  socket.close()
                  // cleanup . . . .
              }
                 catch (Exception e) {
                 // deal with error
              }

           }).start()
       }
    }

  } // class CursorOnTargetSender
```

Next, *sendCoT()* calls the function *buildCoT()* to build an XML *event* document. That function uses the standard Java XML *DocumentBuilderFactory* class and factory that is bundled with every Java based platform (e.g. Android). The document root *event* element is created and configured with the required fields outlined in the previous section. Next the *point* element is created, configured, and added as a child element of the root *event* element. Once the document is complete, it is serialized using a *TransformerFactory* class and a string of characters containing the event message is returned.

Next, *sendCoT()* sends the CoT event message is sent "over the wire" by invoking the *deliverUDP()* method. This function creates and starts a *Runnable* thread that then acquires a multicast UDP socket and sends the CoT the serialized *event* message. Because Android's security environment is a bit more free-wheeling than iOS, no special permission from Google is required to send multicast packets however *android.permission.CHANGE_WIFI_MULTICAST_STATE* must be placed in the application's manifest before compilation.

In order to make the Java code even more portable, Theta has taken a copy of *CursorOnTargetSender.java* from *OpenAthenaAndroid,* stripped it of Android-specific code, and added it to its *ThetaTools* GitHub repository. To compile and run the program, run the following commands from a shell prompt.

```
% javac CursorOnTargetSender.java
% java CursorOnTargetSender -20.0 80.0 120.0
```

The first command compiles the class and the second command runs the program sending a CoT *event* message containing the lat,lon,alt coordinates of -20.0, 80.0 at 120 meters. You can easily modify *sendCoT()* to put in your own event parameters based on your application needs.

Swift/iOS Code Base

Theta's Swift/iOS implementation of CoT can be found in the file *CursorOnTargetSender.swift* within the *OpenAthenaIOS* GitHub repo. The overall code structure roughly parallels the Android version. An instance of the *CursorOnTargetSender* class is first created and then the main method (or function) *sendCoT()* is invoked with the same arguments as its Java/Android counterpart. This function creates an XML CoT *event* message and then sends the message. The caller need not do anything else.

```
public class CursorOnTargetSender
{

  init(params . . .)
  {
    // setup networking, group endpoint
    guard let multicast = try? NWMulticastGroup(for: [ .hostPort(host: host, port: port) [) else {
      return
    }
    .....
    group = NWConnectionGroup(with: multicast, using: parameters)
    group?.stateUpdateHandler = { ..... }
```

```
            group?.setReceiveHandler(.....)
            group?.start(queue: .main)
        }

        deinit()
        {
            group?.cancel()
        }

        public func sendCoT(. . . . .) -> Bool
        {
            // check, validate args
            . . . . .
            // calculate 5-minute interval
            . . . . .

            // build the xml document directly as a string
            var xmlString = "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>"
            xmlString += "<event version=\"2.0\" uid=\"\(uidStr)\" type=\"a-p-G\" how=\"h-c\" "
            xmlString += "time=\"\(exifDateTime)\" start . . . .>"

            // finish building string
            . . . . .

            // send away!
            let sendContent = Data(xmlString.utf8)
            group?.send(content: sendContent) { (error) in
                if error == nil {
             }
                else {
                }
             }

            return true
        }

    } // class CursorOnTargetSender
```

Because Apple XML document classes are available only for MacOS, and because we did not want to import yet another library or Cocoa Pod, the serialized XML CoT *event* message is directly created/coded using builtin Swift language constructs for manipulating string data types. The format of the resulting string is identical to the Java/Android version.

In *OpenAthena* for iOS, the event identifier is created by combining the prefix *openathena-ios* with a platform specific globally unique identifier plus the raw time, in milliseconds, from January 1st, 1970 (a standard UNIX convention for dates and time values). All other CoT *event* message parameters track their Android counterparts.

The CoT *event* message is then sent using the Apple *NWConnectionGroup* libraries provided by iOS. This library, like its Android counterparts, also sends messages asynchronously thus allowing the original calling thread to avoid blocking. Access to multicast messaging, however, is tightly controlled in iOS and requires the developer to request, and receive, an *com.apple.developer.networking.multicast* entitlement before a published application can use this functionality. How this entitlement is obtained and added to the application development environment within *Xcode* is beyond the scope of this document. A simple web search will return multiple articles detailing how to do this.

Python Code Base

Theta thoroughly tested its CoT implementations against both A*TAK* and *iTAK*. During development, however, both A*TAK* and *iTAK* operated as black boxes essentially with very limited error and logging feedback. Thus, if CoT messages were malformed, incorrect, or invalid, *iTAK* and A*TAK* would silently drop them. In order to debug the initial implementations, Theta developed Python UDP multicast sender and receiver applications that produce a lot more debugging information.

The Python programs *udpmcastserver.py* and *udpmcastclient.py* can be used to send and receive test CoT messages and have been tested on Linux and MacOS. (Presumably, they both should work on Windows as well.) The server test program is invoked with a port number and the IP address of an underlying network interface to attach the multicast group membership to. Recall that the default, defacto IPv4 multicast group of 239.2.3.1 is used by TAK. On most UNIX networking protocol stacks, a multicast group must be bound to an underlying network interface and network interfaces are usually specified by their unicast IPv4 address. Once the group is joined, and bound to an underlying interface, the server program loops forever printing out messages/packets it receives. Received CoT messages can then be syntactically and semantically evaluated for correctness.

The client test program, *udpmcastclient.py*, was written to essentially validate that the *iTAK* and A*TAK* programs were correctly operating in the local test environment. This client program obtains a network socket, configures it for multicast, binds it to an underlying interface, and then sends any string the user feeds it. Thus, if a serialized CoT *event* message is sent, and then received by *iTAK* or A*TAK*, the test environment is known to work. Further debugging of new CoT code can progress.

References

1. Theta Limited, https://theta.limited
2. ThetaTools, Theta Limited Repository at GitHub, https://github.com/Theta-Limited/ThetaTools
3. XML, Wikipedia, https://en.wikipedia.org/wiki/XML
4. TAK Specification, The Mitre Corporation, https://www.mitre.org/sites/default/files/pdf/09_4937.pdf
5. Developing Code for IP Multicast on iOS, Apple Inc, https://developer.apple.com/news/?id=0oi77447
6. aTAK, Google Play Store, https://play.google.com/store/apps/details?id=com.atakmap.app.civ
7. iTAK, Apple App Store, https://apps.apple.com/us/app/itak/id1561656396
8. OpenAthenaAndroid Source Code, Theta Limited Repository at GitHub, https://github.com/Theta-Limited/OpenAthenaAndroid
9. OpenAthenaiOS Source Code, Theta Limited Repository at GitHub, https://github.com/Theta-Limited/OpenAthenaIOS
10. Team Awareness Kit (TAK), United States Government, https://tak.gov/
11. CoT XML Schema Definitions, GitHub Repository, https://github.com/mdudel/CoTreceiver/tree/master/lib/xsd
12. TAK Product Center, GitHub Repository, https://github.com/TAK-Product-Center/