HawkVision
Fault-tolerant persistent interactive map
Protocol Design and Specification
March 7th, 2005

Advanced Wireless (600.647) Semester Project
Johns Hopkins University
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Connection Management</td>
<td>4</td>
</tr>
<tr>
<td>Packet Specification</td>
<td></td>
</tr>
<tr>
<td>Packet Field</td>
<td>5</td>
</tr>
<tr>
<td>Packet Type</td>
<td>6</td>
</tr>
<tr>
<td>Data Transmission</td>
<td></td>
</tr>
<tr>
<td>Consistency Model</td>
<td>8</td>
</tr>
<tr>
<td>Message Recovery</td>
<td>9</td>
</tr>
<tr>
<td>IM Messenging</td>
<td>9</td>
</tr>
<tr>
<td>Fault Tolerance and Persistence</td>
<td></td>
</tr>
<tr>
<td>Node Crashes</td>
<td>10</td>
</tr>
<tr>
<td>Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>Flow Control</td>
<td>12</td>
</tr>
<tr>
<td>Conclusion</td>
<td>12</td>
</tr>
</tbody>
</table>
Abstract

Message communication
Each client is assigned a unique numeric id prior to connecting to the system. This id is used for both authentication and for uniquely identifying users in a group. Each user also maintains a knowledge matrix for estimating the knowledge of all other user with respect to objects on the map. One row within the matrix is used by the user to store its current knowledge of objects in the system, known as the knowledge vector. Using this information, clients will be able to manage garbage collection and flow control of messages.

In order two communicate, clients communicate information via two out of band channels. By multicasting, clients will transmit current GPS coordinate information, addition of new map objects, deletion of objects, changes to object attributes such as position and captions, recovery of lost packets, and knowledge vector information. A weak consistency model is employed where coordinate information for map objects are not synchronized across objects, but provided with a mechanism for recovery and propagation, we can ensure eventual consistency.

For point-to-point instant messaging, we employ a unicast using TCP to achieve reliability. Since messaging is intolerant to losses, requires stronger guarantees, and is point-to-point, a weak consistency multicast protocol would perform poorly for the task.

Membership change and Reconciliation
There are no explicit mechanisms for membership. Instead, groups are implicitly acknowledged as new clients are recognized when messages are received from them. In addition, objects are discovered only when messages or knowledge vector packets acknowledge their existence. Hence, given this model, a client eventually hears from and therefore acknowledges all members in the group. Thus, when a new client arrives, other trivially learns about its existence when they receive GPS coordinate updates from it, and vice versa. When a client leaves, others eventually mark the client as disconnected when the client has ceased multicasting coordinates.

In order to facilitate recovery and reconciliation in this system, we propose two mechanisms: a lamport timestamp associated with each unique object for versioning and a default client designated by the sender for resending the recovery (both discussed in detail later). Hence, a disconnected client that remerges with the group will multicast request for most current version of all objects in the system from the first client that it learns is connected to the system. In the case when a client misses an update and later learns about the existence of a new object, it will also multicast the request for the latest version from the client it learned from.

Objects in client knowledge vector
Each client has a unique id and a packet sequence number which is incremented on subsequent packet transmission. Hence, when creating an object, the combinations of the creating client id and sequence number form the object id. For each unique object, the client maintains an entry in its knowledge vector along with versioning information about the object.

Recovery
To tolerate client crashes/reboots, we propose a logging mechanism whereby the client will log each map object update or addition to a log file prior to updating it’s in memory knowledge vector. It also write the knowledge vector along with its current sequence number to persistent storage prior to each update to prevent recycling of sequence numbers. On recovery, the client simply processes the log to rebuild the pre-crash state.
I. Introduction

The design document describes in some detail the ‘HawkVision’ interactive map application, which is designed primarily to assist military command and control in urban combat environments. As such, HawkVision provides a resilient protocol whereby the system is resistant to both catastrophic node crashes and is capable of operating in disconnected environments or within multiple network partitions.

The name HawkVision denotes a god’s eye view of the battlefield similar to a Hawk’s view while surveying the territory for prey.

II. Connection Management

HawkVision clients do not rely on explicit methods for group membership nor does it rely on the existence of a group to function. As such, a client will always assume that it is isolated unless it receives packets from other members on the multicast port. Thus, this mechanism is passive in nature where clients eventually learn about other connected clients rather than explicitly from groups, which allows the system to function in disconnected modes and avoid complexity of a group membership protocol.

One crucial assumption we’ve incorporated is the fact that to ensure security, we assume that a list of potential group members are known ahead of time from a user list installed at each client node. Using this list, clients whether updates from a particular client is valid as only those predefined clients are allowed to communicate.

For connecting to an existing group, we shall discuss how the first node starts. When an isolated node is connecting to the Wave Relay network, it does not make any assumptions about the current membership and defaults to the view that it is alone. However, it will not operate entirely in a disconnected fashion as it will continue to disseminate updates via multicast, only no other members are listening. When subsequent nodes start, they proceed as the first node would. However, eventually, it will receive updates from existing clients while listening to the multicast port, which allows the new client to identify new members on the map. On the flip side, existing clients will eventually receive updates or recovery requests from the new client, which in turn allows them to identify the new client on the map. Hence, connections between members are implicitly formed once they hear each other’s packets.

For disconnection, we do not rely purely on the DISCONNECT to notify clients of membership change. Instead, we rely on a timeout whereby if nothing is heard from a client within a pre-defined threshold, they are marked as disconnected. This works because every client must send out periodic updates for their GPS position, which ensures that connected clients won’t be pruned. One last method from which clients can be marked as disconnected prior to timeout is during IM messaging when the sender finds that the destination client is not longer reachable.
III. Packet Specification

Packet Field

**Client Index (CID 4 bytes)** – unique client index that assigned prior to field operations for authentication and member identification purposes (numeric)

**Message Type (Type 2 bytes)** – indicates what type of message a packet contains. The different types are detailed in the packet type section below.

**Sequence (Seq 4 bytes)** – Sequence number for the message from the originating client. The combination of the client index and sequence number form the unique identifier for each message sent and acts as unique id for each object created.

**Creation Time (Time 8 bytes)** – System creation time of a map object as given by the originating client. The timestamp is a time value denoting the local time on the originating client when the message was initially created.

**Object Type (oType 2 bytes)** – Numeric identifier for object type. We will have a predefined set of objects for the application such as vehicles, ground troops, airborne units, etc, that each object may be associated with.

**Version (Ver 2 bytes)** – Version of the object (lamport timestamp). Version is incremented on each modification by a client.

**Caption (Cap 255 bytes)** – Caption associated with an object describing the object’s attributes or status. Limited to 255 characters. Used under two circumstances; One is to describe an object, and the other as a caption to a special announcement object which carries important messages to the entire group.

**Object Name (Name 30 bytes)** – Name of object. Limited to 50 characters.

**GPS Coordinates (GPS 8 bytes)** – Coordinate information consisting of a latitude/longitude pair (floating point tuple).

**Object Deleted (Deleted 1 byte)** – Marks whether an object is deleted or not. Deleted objects can be garbage collected in memory and on persistent storage once knowledge matrix reflects that all clients have seen the deletion update for the object.

**IM Message Len (MsgLen 4 bytes)** – IM text message length in bytes.

**IM Message (Msg variable sized)** – IM text message sent via unicast to the destination client. Sent via TCP to achieve reliability guarantees.
Packet Type

CONNECT – When a client first attempts to join the group (on recovering from a crash or on initiation), and it first learns of one other connected client, it sends this packet to pass along its attributes and implicitly requests that the connected client to send updated information about all objects (this packet is identical to the UPDATE_OBJECT packet described later except the message type is CONNECT. This simplifies implementation, although DELETED is unnecessary).

<table>
<thead>
<tr>
<th>Type 2 bytes</th>
<th>CID 4 bytes</th>
<th>Seq 4 bytes</th>
<th>Ver 2 bytes</th>
<th>oType 2 bytes</th>
<th>Time 8 bytes</th>
<th>Name 30 bytes</th>
<th>Cap 255 bytes</th>
<th>GPS 8 bytes</th>
<th>Deleted 1 byte</th>
<th>CID (target) 4 bytes</th>
</tr>
</thead>
</table>

DISCONNECT – a packet from the client to explicitly disconnect from the group. Although not necessary, it serves to speed up the propagation of the client’s disconnected state in order for other clients to reassign ignore its knowledge vector entries in future.

<table>
<thead>
<tr>
<th>Type 2 bytes</th>
<th>CID 4 bytes</th>
<th>Seq 4 bytes</th>
<th>Ver 2 bytes</th>
<th>GPS 8 bytes</th>
</tr>
</thead>
</table>

CLIENT_STATE – a packet send periodically from the client containing the most current GPS positioning information. In addition, it contains a vector of objects that the client currently believes are not up-to-date among all other connected clients, including the lamport timestamp/version info (note, we might improve the efficiency of the system later by not sending the vector with each packet, and instead, include them at fixed intervals). Note that objects are included in the vector based on the knowledge matrix at the client, which indicates what objects are not consistent among other clients and for which the current client has the most up-to-date version.

<table>
<thead>
<tr>
<th>Type 2 bytes</th>
<th>CID 4 bytes</th>
<th>Seq 4 bytes</th>
<th>Ver 2 bytes</th>
<th>GPS 8 bytes</th>
<th>Vector Size 2 bytes</th>
<th>Object Vector (variable sized) – Vector of (CID, Seq, Ver) items Each 10 bytes in size.</th>
</tr>
</thead>
</table>

UPDATE_OBJECT – a packet from a client either creating a new object or modifying an existing object (create, delete, update). For objects that represent the clients, these are special objects that always have a sequence number of 0. Also only a client can update itself (target CID is unnecessary but simplifies implementation).

<table>
<thead>
<tr>
<th>Type 2 bytes</th>
<th>CID 4 bytes</th>
<th>Seq 4 bytes</th>
<th>Ver 2 bytes</th>
<th>oType 2 bytes</th>
<th>Time 8 bytes</th>
<th>Name 30 bytes</th>
<th>Cap 255 bytes</th>
<th>GPS 8 bytes</th>
<th>Deleted 1 byte</th>
<th>CID (target) 4 bytes</th>
</tr>
</thead>
</table>
KNOWLEDGE – knowledge vector containing the list of objects known to a given client from the map. This information is sent out periodically over long intervals to facilitate bringing group members up-to-date. Information from the vector help client identify existing objects and latest version number, allowing clients to detect when safe deletion of object is allowed and which objects are missing or out-of-date in its view.

<table>
<thead>
<tr>
<th>Type 2 bytes</th>
<th>CID 4 bytes</th>
<th>Seq 4 bytes</th>
<th>Ver 2 bytes</th>
<th>Vector Size 2 bytes</th>
<th>Knowledge Vector (variable sized) – Vector of (CID, Seq, Ver) items Each 10 bytes in size</th>
</tr>
</thead>
</table>

RECOVERY_REQUEST – a packet containing a request for attributes of the object specified in the event that a client discovers a new object exists but did not receive the update packet with respect to the object. The client specified in target CID (ie. the client that originally notified the sender of the missing update) will respond with the object attribute (the CID, Seq, and Ver combination uniquely identifies the object and version that is requested). If another client receives this request, has the latest version and marked the target CID as disconnected, then it will respond instead.

<table>
<thead>
<tr>
<th>Type 2 bytes</th>
<th>CID 4 bytes</th>
<th>Seq 4 bytes</th>
<th>Ver 2 bytes</th>
<th>CID (target) 4 bytes</th>
</tr>
</thead>
</table>

IM_MESSAGE – a packet containing a point-to-point IM message between connected clients (obviously, communication is only possible when the receiving client has been discovered by the sender).

<table>
<thead>
<tr>
<th>Type 2 bytes</th>
<th>CID 4 bytes</th>
<th>Seq 4 bytes</th>
<th>Ver 2 bytes</th>
<th>Time 8 bytes</th>
<th>MsgLen 4 bytes</th>
<th>Msg – variable sized</th>
</tr>
</thead>
</table>

COMPROMISED – packet from a client informing others that the system is compromised, which will cause all nodes to initiate a form of self-destruct sequence (this can be done when a client discovers that a node has been captured by the enemy). The details of this mechanism will be discussed in detail during late iterations.

<table>
<thead>
<tr>
<th>Type 2 bytes</th>
<th>CID 4 bytes</th>
<th>Seq 4 bytes</th>
<th>Ver 2 bytes</th>
<th>Time 8 bytes</th>
</tr>
</thead>
</table>
IV. Data Transmission

Consistency Model

Multicast messages are used to carry updates for interactive map objects to all clients in the system. Although some messages may be lost in the process, using the recovery mechanism discussed next, all updates eventually propagate through the system.

We have only one type of update packet, namely UPDATE_OBJECT. This packet captures all required information to represent all map objects. Each object has a unique object ID which is the client ID and sequence number pair. For the client node objects, their ID is simply the client ID and a sequence number of zero. For all other objects such as landing zones, buildings, targets, etc, the object ID is the client ID and sequence number assigned by the creating client. Finally, a special global announcement message object is used to carry announcement broadcasts to the entire group and is identified by having both client ID and sequence number being zero. Hence, each object created can be identified uniquely.

In order to propagate a consistent version of an object through the system, we introduce a lamport time stamp for each object as a crude versioning mechanism. When a client first creates an object, he assigns a name, description, coordinate, and type associated with the object and multicasts this information to the group while assigning it a version number of 1. When other clients need to make a change to the object, they simply increment the object’s version number and multicasts the update so other clients may replace their current view of the object (clients can also rely on the version number to discard older updates of the same object). Of course, this mechanism may result in version collisions in which multiple clients make the changes to the same base version resulting in duplicate inconsistent views of the object. However, this collision should occur rarely and does not represent a major flaw as subsequent updates to the object will result in a consistent version.

Also, concerning what constitutes a version change, we specify that for objects created by the clients, who are presumably fairly static relative to their GPS coordinates, a version change is warranted when a client changes any aspect of the object, including its name, description, coordinates, etc. However, for mobile client nodes operating the HawkVision unit, we will assume that they are non-static and will multicast positional updates periodically. As such, enforcing a version change for every GPS update would unduly impact network performance. Thus, coordinate updates for clients does not cause a version change since updates are sent periodically and losing one particular update from a client does not represent a hindrance so long as network quality is reasonable.

Hence, a combination of unique object ID and versioning system allow updates to propagate through the system and ensure that client view of the interactive map reach eventual consistency.
Message Recovery

Invariably, since we are not building a reliable group communication protocol, object updates will get lost and a recovery mechanism must be in place. We employ a lazy pull-centric approach whereby, a client does not attempt to recover an update until it “learns” about a new version of an object missing from its view. We will first describe how a client learns about new objects and then how those objects are recovered.

As described in the packet types section above, two types of message carry knowledge information from other clients, namely KNOWLEDGE and CLIENT_STATE packets. These packets each contain a vector condensing the knowledge of a client, namely a list of object ID in its view along with the current version number. Comparing knowledge of other against ones own, a client can detect creation of new objects or an object version that is out-of-date.

Once a client detects a missing object update, it will proceed to multicast RECOVERY_REQUEST packets for each object it missed in regular intervals until objects are recovered. To determine which client will respond to the request, we introduce the concept of target CID below.

A client learns of a missing update from the CLIENT_STATE or KNOWLEDGE packet of at least one client, which we call the target CID. Thus, the client requesting the missing update will attached a target CID to the RECOVERY packet since it knows that the target client is guaranteed to have a newer or equal version. So, when a client receives a recovery request, if it has the latest version of the object requested and it has the target CID, then it will multicast the object’s update (Consequently, the implications for deleting an object is that objects are kept around on storage and in memory until a client knows that all other clients, from a predefined list, have seen the deletion update. This knowledge is gained from KNOWLEDGE packets). However, if the target client is disconnected, then another client that receives the request, has a newer or equal version, and marked the target client as disconnected, will respond with the update instead.

IM Messaging

Regarding point-to-point instant messaging, we rely on TCP to ensure reliable delivery of messages. Obviously, a client can only initiate an IM transmission to clients that it received an update packet from recently (ie. connected clients). For clients that are marked are connected, the sender maintains a list of these clients along with their IP address extracted from the last packet received from the client (hence, every node maintains two listener threads, one for multicast updates and one for unicast IM messages). When the applications delivers a message to the network layer, we look-up the IP and then send the message using TCP. TCP will ensure that the message arrives at the client and no further steps are taken.
V. Fault Tolerance and Persistence

Node Crashes

In order to ensure that nodes can sustain random crashes resulting from either physical damage unforeseen software glitch, we design a mechanism to ensure high availability by writing updates to persistent storage. Once the device is repaired and recovers from a crash, it can continue to function normally.

As discussed before, allowing disconnected operations is one form of high availability. We will also assume that the device is attached to a removable persistent storage medium preferably a tape device. On the device, we will sequentially log updates that can be replayed on another unit to recover the pre-crash state. The logging will consist of two pieces of information: the last used sequence number and the list of updated objects along with their current attributes.

The sequence number is written to disk immediately before use to prevent recycling of the sequence number. This is crucial as an object’s unique identifier depends on the sequence number and thus on recovery from crashes, the client must not reuse past sequence numbers, which would result in different objects sharing the same identifier.

We also log the client’s view or knowledge vector to disk containing a list of known objects along with their current attributes. Object updates are logged immediately upon reception and updates being sent are logged prior to transmission to the multicast group. By replaying the log, a client can recover its knowledge prior to crashing. In addition, the knowledge matrix containing the knowledge of all other clients (i.e. the most current version of objects known to others), is logged after each object update.

‘With the above logging mechanism in mind, a client that attempts to rejoin the group will only request recovery for object updates since its last crash. It simply replays the log to recover its state and blocks updates until log recovery is complete.

**Algorithm (pseudo-code: network layer for client)**

*On Start*
Process log from persistent storage
- Recover last sequence number
- Restore knowledge vector (all known object updates)

Send CONNECT message

*Normal Operation*

*While True*

If position update threshold
- Overwrite coordinates with last position from GPS device
- Multicast CLIENT_STATE with current coordinates. Also check knowledge matrix for all objects for which at least one client might not
have an up-to-date version and the current client has the most up-to-date version. For these objects, attach their ID and version to the packet as a vector.

If received multicast packet

If DISCONNECTED message
- mark client as disconnected in memory (also ignores knowledge vector entry for disconnected client in knowledge matrix).

If CLIENT_STATE or KNOWLEDGE message
- update coordinates corresponding to client
- Update knowledge vector entry in own knowledge matrix corresponding to the sending client and log update
- Comparing sender’s view with own knowledge vector. If objects are out-of-date or does not exist in own view, queue recovery request for the missing update, addressed to the sender of the message, to recovery queue (sent periodically).

If UPDATE_OBJECT message
- compare to current version of object, if newer version or new object, insert/update knowledge vector and log update
- Compare object to recovery queue, if found versions that are equal or older on the queue, remove recovery requests for object from queue

If RECOVERY_REQUEST message
- Extract object ID, version, and designated client ID from message
- If current client has a newer or equal version and is designated by the sender as the target for resending the missing update, then resend it
- Else, if current client has a newer or equal version and the designated target is known to be disconnected, then resend the update

If received unicast packet
- Examine sender ID for IM message
- If sender is marked as connected (ie, recently sent GPS update), deliver IM message to application

If pruning threshold for GPS updates reached
- For each client, check if GPS updates received within last period
- If no updates recently, mark client as disconnected.
- Reset GPS update counter for all connected clients
- Multicast KNOWLEDGE containing knowledge vector
VI. Flow Control

Since this system relies heavily on group multicast with periodic GPS coordinate updates, improperly handling updates and recovery could result in poor flow control and degraded performance. We envision that the system will operate at a maximum of a few dozen nodes hence we will need to tweak the GPS position update interval to ensure that the positioning update overhead is low.

With regards to multicast updates, our system only multicasts update only once, regardless of whether it is received. Hence it is optimal. When recovering, only the client that lost the update and later discovers the update will send requests for the packet, but since the recovery responsibility defaults to the client that notified the sender of the missing update, we ensure that only one redundant packet is sent for each request (the case in which the target client disconnects and other clients take over the responsibility of responding to the request should be rare, and redundant transmission will occur it that scenario). Furthermore, the problem arises when multiple clients send recovery requests for the same object in succession due to high loss network. The worst case scenario is when a target client processes these requests in rapid succession and resends the update multiple times when only one additional retransmission is required. To alleviate this problem, we borrow from the fact that a multicast channel is used, hence a recovered update for one client is likely to be useful to other clients that missed the same update. Hence, for each recently requested object, the client maintains a counter that if rapid successive requests arrive for the same object, the client ignores additional requests as long as the counter has not expired.

Finally, we argue that object updates, which consume a majority of the bandwidth since they carry large amounts of data relative to other packets, are transmitted in a near optimal fashion and are relatively static so as to not require many additional updates once the object is created. The only exception is the mobile clients themselves, in which we make an exception but not triggering a version change for these clients for positional updates, hence reducing the need to retransmit full attributes with each GPS update.

VII. Conclusion

In summary, we rely on several mechanisms involving lamport time stamp, targeted recovery, knowledge vector, and others to ensure a weak consistency update model that allows clients to maintain a relatively consistent view of the map with mechanisms for eventual recovery and convergence. We also enforce logging to protect against unforeseen crashes. As a result, HawkVision is capable of providing a robust, fault tolerant, and high availability service resistant to disconnections and physical crashes.