Network Tables 2.0
Protocol Proposal
7/9/2012
Protocol Revision 0.0

Table of Contents

Introduction

This document defines a network protocol for a key-value store that may be read from and written to by multiple remote clients. A central server, most often running on a FIRST FRC robot controller, is responsible for providing information consistency and for facilitating communication between clients.

Information consistency is guaranteed through the use of a sequence number associated with each key-value pair. An update of a key-value pair increments the associated sequence number, and this update information is shared with all participating clients. The central server only applies and redistributes updates which have a larger sequence number than its own, which guarantees that a client must have received a server's most recent state before it can replace it with a new value.

The concept of a transaction is also provided; transactions allow a sender to specify that a group of updates be applied at the same time. More specifically, user code running on a the receiving device must perceive the group of updates as occurring simultaneously, regardless of network I/O or thread scheduling delays.

This is a backwards-incompatible rework of the Network Tables network protocol originally introduced for the 2012 FIRST Robotics Competition. Note that this revision of the Network Tables protocol no longer includes the concept of sub-tables. We suggest that instead of representing sub-tables as first-class data types in the network protocol, it would be easy for an implementation to provide a similar API abstraction by adding prefixes to keys. For example, we suggest using Unix-style path strings to define sub-table hierarchies. The prefix ensures that sub-table namespaces do not collide in a global hashtable without requiring an explicit sub-table representation.

Definitions

Client - An implementation of this protocol running in client configuration. Any number of Clients may exist for a given Network.

Entry - A data value identified by a string name.

Entry ID - An unsigned 2-byte ID by which the Server and Clients refer to an Entry across the network instead of using the full string key for the Entry. Entry IDs range from 0x0000 to 0xFFFE (0xFFFF is reserved for an Entry Assignment issued by a Client).

Server - An implementation of this protocol running in server configuration. One and only one Server must exist for a given Network.

Network - One or more Client nodes connected to a Server.

User Code - User-supplied code which may interact with a Client or Server. User Code should be executed on the same computer as the Client or Server instance it interacts with.

Sequence Number - An unsigned number which allows the Server to resolve update conflicts between Clients and/or the Server. Sequence numbers may overflow.

Sequential arithmetic comparisons, which must be used with Sequence Numbers, are defined by RFC 1982.

This document conforms to RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels.

Protocol

Transport Layer

Conventional implementations of this protocol should use TCP for reliable communication; the Server should listen on TCP port 1735 for incoming connections.

Example Exchanges

Client Connects to the Server

Directly after client establishes a connection with the Server, the following procedure must be followed:

1. The Client sends a Client Hello message to the Server
2. The Server sends a Begin Transaction message.
3. The Server sends one Entry Assignment for every field it currently recognizes.
4. The Server sends an End Transaction message.
5. For all Entries the Client recognizes that the Server did not identify with a Entry Assignment, the client follows the Client Creates an Entry protocol.

In the event that the Server does not support the protocol revision that the Client has requested in a Client Hello message, the Server must instead issue a Protocol Version Unsupported message to the joining client and close the connection.

Client Creates an Entry

When User Code on a Client assigns a value to an Entry that the Server has not yet issued a Entry Assignment for, the following procedure must be followed:

1. The Client sends an Entry Assignment with an Entry ID of 0xFFFF.
2. The Server issues an Entry Assignment to all Clients (including the sender) for the new field containing a real Entry ID and Sequence Number for the new field.

In the event that User Code on the Client updates the value of the to-be-announced field again before the expected Entry Assignment is received, then the Client must save the new value and take no other action (the most recent value of the field should be issued when the Entry Assignment arrives, if it differs from the value contained in the received Entry Assignment).

In the event that the Client receives a Entry Assignment from the Server for the Entry that it intended to issue an Entry Assignment for, before it issued its own Entry Assignment, the procedure should end early.

In the event that the Server receives a duplicate Entry Assignment from a Client (likely due to the client having not yet received the Server's Entry Assignment), the Server should ignore the duplicate Entry Assignment.

Client Updates an Entry

When User Code on a Client updates the value of an Entry, the Client must send an Entry Update message to the Server. The Sequence Number included in the Entry Update message must be the most recently received Sequence Number for the Entry to be updated incremented by one.

Example:

1. Client receives Entry Assignment message for Entry "a" with integer value 1, Entry ID of 0, and Sequence Number 1.
2. User Code on Client updates value of Entry "a" to 16 (arbitrary).
3. Client sends Entry Update message to Server for Entry 0 with a Sequence Number of 2 and a value of 16.

When the Server receives an Entry Update message, it first checks the Sequence Number in the message against the Server's value for the Sequence Number associated with the Entry to be updated. If the received Sequence Number is strictly greater than (aside: see definition of "greater than" under the definition of Sequence Number) the Server's Sequence Number for the Entry to be updated, the Server must apply the new value for the indicated Entry and repeat the Entry Update message to all other connected Clients.

If the received Sequence Number is less than or equal (see definition of “less than or equal” in RFC 1982) to the Server’s Sequence Number for the Entry to be updated, this implies that the Client which issued the Entry Update message has not yet received one or more Entry Update message(s) that the Server recently sent to it; therefore, the Server must ignore the received Entry Update message. In the event that comparison between two Sequence Numbers is undefined (see RFC 1982), then the Server must always win (it ignores the Entry Update message under consideration).

Implementation Note: If User Code modifies the value of an Entry too quickly, 1) users may not see every value appear on remote machines, and 2) the consistency protection offered by the Entry’s Sequence Number may be lost (by overflowing before remote devices hear recent values). It is recommended that implementations detect when user code updates an Entry more frequently than once every 5 milliseconds and print a warning message to the user (and/or offer some other means of informing User Code of this condition).

Server Creates an Entry

When User Code on the Server assigns a value to a Entry which does not exist, the Server must issue an Entry Assignment message to all connected clients.

Server Updates an Entry

When local User Code updates the value of an Entry on the Server, the Server must apply the new value to the Entry immediately, increment the associated Entry’s Sequence Number, and issue a Entry Update message containing the new value and Sequence Number of the associated Entry to all connected Clients.

Implementation Note: See Implementation Note under Client Updates an Entry above.

Transactions

Transactions guarantee that a set of Entry Assignments and/or Entry Updates is applied on a remote device all at once from the perspective of User Code running on the remote device. The results of operations contained within a received transaction must not be visible to User Code until the entire transaction has been received and processed.

To issue a transaction:

1. The originating node (i.e. a Client or the Server) sends a Begin Transaction message
2. The originating node sends one or more Entry Assignments and/or one or more Entry Updates
3. The originating node sends an End Transaction message

Keep Alive

To maintain a connection and prove a socket is still open, a Client or Server may issue Keep Alive messages. Clients and the Server should ignore incoming Keep Alive messages.

The intent is that by writing a Keep Alive to a socket, a Client forces its network layer (TCP) to reevaluate the state of the network connection as it attempts to deliver the Keep Alive message. In the event that a connection is no longer usable, a Client’s network layer should inform the Client that it is no longer usable within a few attempts to send a Keep Alive message.

To provide timely connection status information, Clients should send a Keep Alive message to the Server after every 1 second period of connection inactivity (i.e. no information is being sent to the Server). Clients should not send Keep Alive messages more frequently than once every 100 milliseconds.

Since the Server does not require as timely information about the status of a connection, it is not required to send Keep Alive messages during a period of inactivity.

Bandwidth and Latency Considerations

To reduce unnecessary bandwidth usage, implementations of this protocol should:

- Send an Entry Update if and only if the value of an Entry is changed to a value that is different from its prior value.
- Buffer messages and transmit them in groups, when possible, to reduce transport protocol overhead.
- Only send the most recent value of an Entry. When User Code updates the value of an Entry more than once before the new value is transmitted, only the latest value of the Entry should be sent.

It is important to note that these behaviors will increase the latency between when a Client or Server updates the value of an Entry and when all Clients reflect the new value. The exact behavior of this buffering is left to implementations to determine, although the chosen scheme should reflect the needs of User Code. Implementations may include a method by which User Code can specify the maximum tolerable send latency.

Message Structures

All messages are of the following format:

Field Name	Field Type
Message Type	1 byte, unsigned
Message Data	N bytes (length determined by Message Type)

Keep Alive

Indicates that the remote party is checking the status of a network connection.

Field Name	Field Type
0x00 - Keep Alive Message	1 byte, unsigned; Message Type

Client Hello

A Client issues a Client Hello message when first establishing a connection. The Client Protocol Revision field specifies the Network Table protocol revision that the Client would like to use.

Field Name	Field Type
0x01 - Client Hello Message	1 byte, unsigned; Message Type
Client Protocol Revision	Unsigned 16-bit integer (big-endian)

Protocol Version Unsupported

A Server issues a Protocol Version Unsupported message to a Client to inform it that the requested protocol revision is not supported. It also includes the most recent protocol revision which it supports, such that a Client may reconnect under a prior protocol revision if able.

Field Name	Field Type
0x02	1 byte, unsigned; Message Type
Server Supported Protocol Revision	Unsigned 16-bit integer (big-endian)

Entry Assignment

A Entry Assignment message informs the remote party of a new Entry. An Entry Assignment's value field must be the most recent value of the field being assigned at the time that the Entry Assignment is sent.

Field Name	Field Type
0x10 - Entry Assignment Message	1 byte, unsigned; Message Type
Entry Name	String; see Entry Value definition
Entry Type	1 byte, unsigned
Entry ID	2 byte, unsigned
Entry Sequence Number	2 bytes, unsigned
Entry Value	N bytes, length depends on Entry Type

If the Entry ID is 0xFFFF, then this assignment represents a request from a Client to the Server. In this event, the Entry ID field and the Entry Sequence Number field must not be stored or relied upon as they otherwise would be.

Entry Type must assume one the following values:

0x00	Boolean
0x01	Double

0x02	String

Entry Value must assume the following structure as indicated by Entry Type:

Entry Type	Entry Value Format
Boolean	1 byte, unsigned; True = 0x01, False = 0x00
Double	IEEE 754 floating-point "double format" bit layout; (big endian)
String	2 byte, unsigned length prefix (big endian), followed by Java modified UTF-8 format

Entry Update

An Entry Update message informs a remote party of a new value for an Entry.

Field Name	Field Type
0x11 - Field Update	1 byte, unsigned; Message Type
Entry ID	2 byte, unsigned
Entry Sequence Number	2 bytes, unsigned
Entry Value	N bytes, length dependent on value type

Begin Transaction

A Begin Transaction message informs the remote party that the following Entry Assignments and/or Entry Updates must be applied at the same time from the perspective of User Code.

Field Name	Field Type
0x20	1 byte, unsigned; Message Type

End Transaction

An End Transaction message informs the remote party that a transaction previously opened with a Begin Transaction message has been fully received and may be processed.

Field Name	Field Type
0x21	1 byte, unsigned; Message Type

Appendix A - Alloy Model

Alloy (http://alloy.mit.edu/alloy/) is a formal logic tool that can analyze first-order logic expressions. Under the proposed sequence number -based protocol, assuming that all nodes start from the same state, Alloy is unable to find a way where two nodes with the same sequence number have different state when activity ceases.

The model I used is included below. Although Alloy cannot test all cases, since such an exhaustive search is intractable, it provides a high level of confidence in the proposed protocol.

```
--- Models a distributed, centralized hash table system called
NetworkTables
--- System state is protected by sequence numbers; the server's value
for a certain sequence number always wins
--- Paul Malmsten, 2012 pmalmsten@gmail.com

open util/ordering[Time] as TO
open util/natural as natural

sig Time {}
sig State {}

--- Define nodes and server
sig Node {
    state: State -> Time,
    sequenceNumber: Natural -> Time
}

--- Only one server
one sig Server extends Node {
}

--- Define possible events
abstract sig Event {
     pre,post: Time,
     receiver: one Node
}

// For all events, event.post is the time directly following event.pre
fact {
     all e:Event {
          e.post = e.pre.next
     }
}

// Represents that state has changed on a node
sig StateChangeEvent extends Event {
}

// Represents that state has been transferred from one node to another
```

```
sig StateTransferEvent extends Event {
      sender: one Node
}

fact {
      --- Every node must assume at most one state
      all t:Time, n:Node | #n.state.t = 1

      --- Every node must assume one sequence number
      all t:Time, n:Node | #n.sequenceNumber.t = 1

      --- Sequence numbers may only increment
      all t:Time - last, n:Node | let t' = t.next | natural/
gte[n.sequenceNumber.t', n.sequenceNumber.t]
}


fact stateChangedImpliesAStateTransfer {
      all sce:StateChangeEvent {
            // A StateChange on a client causes a transfer to the
Server if its sequence number is greater than the server's
            sce.receiver in Node - Server and natural/
gt[sce.receiver.sequenceNumber.(sce.post),
Server.sequenceNumber.(sce.post)]
             implies
                  some ste:StateTransferEvent {
                        ste.pre = sce.post and ste.sender = sce.receiver
and ste.receiver = Server
                  }
      }

      all sce:StateChangeEvent {
            // A StateChange on the server causes a transfer to all
clients
            sce.receiver = Server implies
                  all n:Node - Server {
                        some ste:StateTransferEvent {
                              ste.pre = sce.post and ste.sender =
Server and ste.receiver = n
                        }
                  }
      }

      all sce:StateTransferEvent {
            // A StateTransfer to the server causes a transfer to all
clients
            sce.receiver = Server implies
                  all n:Node - Server {
                        some ste:StateTransferEvent {
                              ste.pre = sce.post and ste.sender =
Server and ste.receiver = n
```

```
                    }
                }
        }
}

fact stateTransferEventsMoveState {
        all ste:StateTransferEvent {
                ste.sender = Server and not ste.receiver = Server or
ste.receiver = Server and not ste.sender = Server

                // Nodes can only post to the server if their sequence
number is greater than the servers
                ste.receiver = Server implies natural/
gt[ste.sender.sequenceNumber.(ste.pre),
ste.receiver.sequenceNumber.(ste.pre)]

                // Server can only post to clients if its sequence number
is greater than or equal to the client
                ste.sender = Server implies natural/
gte[ste.sender.sequenceNumber.(ste.pre),
ste.receiver.sequenceNumber.(ste.pre)]

                // Actual transfer
                (ste.receiver.state.(ste.post) = ste.sender.state.(ste.pre)
and
                  ste.receiver.sequenceNumber.(ste.post) =
ste.sender.sequenceNumber.(ste.pre))
        }
}


fact noEventsPendingAtEnd {
        no e:Event {
                e.pre = last
        }
}

fact noDuplicateEvents {
        all e,e2:Event {
                // Two different events with the same receiver imply they
occurred at different times
                e.receiver = e2.receiver and e != e2 implies e.pre != e2.pre
        }
}

fact noStateTransfersToSelf {
        all ste:StateTransferEvent {
                ste.sender != ste.receiver
        }
}

fact noDuplicateStateTransferEvents {
```

```
	all ste,ste2:StateTransferEvent {
		// Two state transfer events with the same nodes imply that
they occurred at different times
		ste.sender = ste2.sender and ste.receiver = ste2.receiver
and ste != ste2 implies ste.pre != ste2.pre
	}
}

--- Trace (time invariant)
fact trace {
	all t:Time - last | let t' = t.next {
		all n:Node {
			// A node in (pre.t).receiver means it is being
pointed to by some event that will happen over the next time step
			n in (pre.t).receiver implies n.state.t' !=
n.state.t and n.sequenceNumber.t' != n.sequenceNumber.t	// A node
which receives some sort of event at time t causes it to change state
else n.state.t' = n.state.t and n.sequenceNumber.t' =
n.sequenceNumber.t	// Otherwise, it does not change state
		}
	}
}

--- Things we might like to be true, but are not always true

pred atLeastOneEvent {
	#Event >= 1
}

pred allNodesStartAtSameStateAndSequenceNumber {
	all n,n2:Node {
		n.state.first = n2.state.first and n.sequenceNumber.first =
n2.sequenceNumber.first
	}
}

pred noStateChangeEventsAtEnd {
	no e:StateChangeEvent {
		e.post = last
	}
}

--- Demonstration (Alloy will try to satisfy this)

pred show {
	atLeastOneEvent
}
run show

--- Assertions (Alloy will try to break these)

assert allNodesConsistentAtEnd {
```

```
      allNodesStartAtSameStateAndSequenceNumber implies
            all n,n2:Node {
                  // At the end of a sequence (last) all nodes with the
same sequence number have the same state
                  n.sequenceNumber.last = n2.sequenceNumber.last
implies n.state.last = n2.state.last
            }
}
check allNodesConsistentAtEnd for 3 Event, 10 Node, 3 State, 5 Time, 5
Natural
check allNodesConsistentAtEnd for 8 Event, 2 Node, 5 State, 9 Time, 9
Natural

assert serverHasHighestSeqNumAtEnd {
      allNodesStartAtSameStateAndSequenceNumber implies
            all n:Node - Server{
                  // At the end of a sequence (last) all nodes with the
same sequence number have the same state
                  natural/gte[Server.sequenceNumber.last,
n.sequenceNumber.last]
            }
}
check serverHasHighestSeqNumAtEnd for 3 Event, 10 Node, 3 State, 5
Time, 5 Natural
```