

The Turtle F2F Client – Developer’s Manual

The Turtle client has been implemented as a modular architecture, as shown in Figure 1.

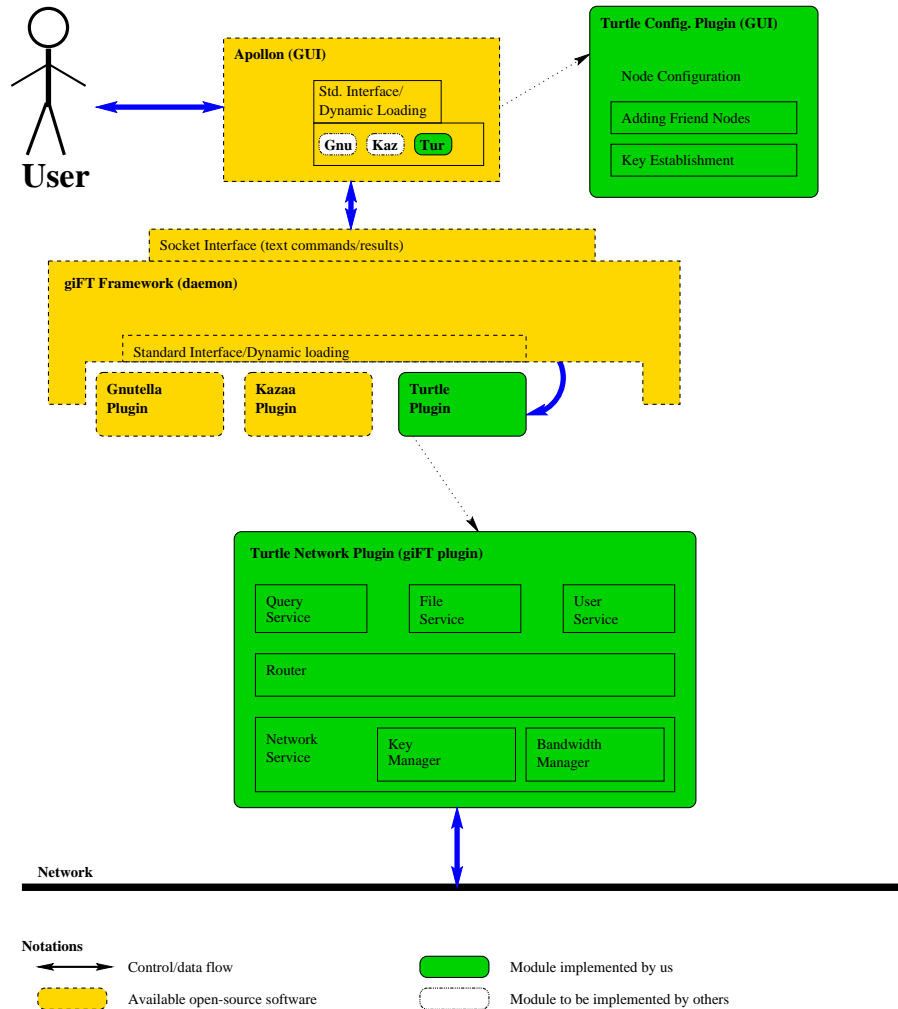


Figure 1. The software structure of the Turtle client

As shown in Figure 1 the Turtle client consists of four main components:

- The Apollon GUI client provides the user front-end. Apollon is third-party, open source software, and can be downloaded from <http://apollon.sourceforge.net/>. Apollon is

written in C++ for the KDE Desktop environment, and makes heavy use of the QT graphic libraries. We provide a brief description of the most important features of the Apollon software architecture in Section 1. The Apollon project Web site contains more detailed information for Apollon developers.

- The Turtle configuration plugin for Apollon provides a GUI front-end for configuring the Turtle protocol settings from Apollon. This plugin allows configuring the Turtle node, adding friend nodes, and establishing shared keys with friends based on common knowledge. We provide a brief description of the software architecture of this plugin in Section 2.
- The giFT daemon is a software component that allows multiple P2P protocols to be executed from the same application. Supported protocols are implemented as giFT plugins. giFT is third-party, open source software, and can be downloaded from <http://gift.sourceforge.net/>. We provide a brief description of the most important features of the giFT software architecture in Section 3. The giFT project Web site contains additional information for giFT developers.
- The Turtle plugin for giFT is the component that implements the actual Turtle protocol. Its software architecture is described in detail in Section 4.

1 The Apollon Client

Apollon is a popular open source P2P client, designed as a GUI front-end for the giFT daemon. Since it has not been developed by us, details about Apollon's software architecture are outside the scope of this document; as a high level description, Apollon is written in C++ for the KDE Desktop environment, and makes heavy use of the QT graphic libraries. More information for Apollon developers can be found on the project Web site—<http://apollon.sourceforge.net/>.

For the Turtle project, we decided to modify Apollon so it can support graphical (“point-and-click”) configuration of Turtle nodes. Our main motivation has been to keep changes in the actual Apollon code minimal. As such, we have decided to implement all our changes as a shared library, which is then loaded by Apollon when Turtle configuration is requested by the user. As such, the actual changes to the Apollon source code are confined in one file—*preferences.ui.h*. Here we have added one extra (internal) function—*createConfigDialog()*. This function does the following:

- Searches the Apollon installation directory for one file—*libTurtleConfigPlugin.so* which contains the shared library implementing the Turtle configuration plugin.
- If the file is found, the function loads the shared library using the *dlopen()* utility.
- If *dlopen()* succeeds, *createConfigDialog()* makes a call to *dlsym()* utility in order to find the *factory()* function exported by the shared library.
- Finally, *createConfigDialog()* calls the *factory()* function which in turn creates the Turtle configuration dialog.

In addition to the *createConfigDialog()* function, we have also added one button—*Configure*—to the *Appollon Preferences* dialog (defined in the *preferences.ui* file). Pressing this button while the Turtle plugin is selected invokes the *createConfigDialog()* function, which in turn creates the Turtle configuration dialog.

2 The Turtle Configuration Plugin for Apollon

The Turtle configuration plugin for Apollon is a GUI front-end for configuring Turtle nodes from within the Apollon client. This plugin has been designed as a shared library (*libTurtleConfigPlugin.so*), which is loaded by the main Apollon program.

The *libTurtleConfigPlugin.so* exports only one function *factory()* (defined in *TurtleConfigReceiver.h* and implemented in *TurtleConfigReceiver.cpp*), which creates the main dialog for configuring Turtle nodes.

The configuration plugin implements three distinct dialogs:

- The Turtle Config Dialog—the main dialog window for configuring the Turtle node. This dialog contains text boxes for entering the Turtle node’s ID, TCP port (where the node listens for connection requests from friend nodes), and so on. This dialog is defined in the *TurtleConfigDialog.ui* file. The functionality of the dialog (e.g. the actions that take place when the various buttons/scrolls are clicked) is implemented in the *TurtleConfigReceiver.cpp* file.
- The Friends Settings Dialog—this dialog is used to add/edit Turtle friend node entries. The dialog contains text boxes for entering the friend node’s ID, IP address, port, shared key, and so on. This dialog is defined in the *FriendSettingsDialog.ui* file. The functionality of the dialog (e.g. the actions that take place when the various buttons/scrolls are clicked) is implemented in the *FriendSettingsControl.cpp* file.
- The Key Agreement Dialog—this dialog implements the interactive key agreement protocol between two friend Turtle nodes. The protocol constructs a secret shared key through an interactive questions/answers session, where each of the two friends asks questions for which both parties know the correct answer (but this answer is not obvious to the rest of the world. The answers to these questions are incorporated (via secure hashing) into the shared key. The key agreement dialog contains text boxes for typing questions and answers, as well as a scrolldown choice list with “standard” questions (for example: “What is your pet’s name?”). This dialog is defined in the *KeyAgreementDialog.ui* file. The functionality of the dialog (e.g. the actions that take place when the various buttons/scrolls are clicked) is implemented in the *KeyAgreementControl.cpp* file. This is probably the most complex file in the entire plugin, since it contains the implementation of the interactive questions/answers protocol state machine.

3 The giFT Daemon

giFT is a software component that allows accessing multiple P2P networks from the same application. In order to accomplish this, giFT is designed as a modular architecture, where components (plugins) implementing different P2P protocols are dynamically loaded. Turtle has been designed as one such giFT plugin.

giFT is third-party, open-source software. As such, the details of the giFT software architecture are outside the scope of this thesis. More details for giFT developers can be found on the project Web site—<http://gift.sourceforge.net/>.

In this document we focus on the part of giFT that deals with loading and initializing protocol plugins. These protocol plugins need to be implemented as shared libraries, and are loaded using the *dlopen()* (this takes place in the *src/plugin.c* file). Shared libraries implementing protocol plugins

should always export exactly one function— $\{Protocol\ Name\}_init(Protocol *p)$ (thus for the Turtle plugin, the name of the exported function is *Turtle_init()*). The init function is passed a pointer to a *Protocol* structure (defined by giFT in *plugin/protocol.h*), which contains all necessary information about the P2P protocol implemented by the library. The most important information in this *Protocol* structure consists of function pointers implementing the plugin's functionality; some of the relevant functions are:

- **start*—called by giFT to start the plugin.
- **destroy*—called by giFT to destroy the plugin.
- **search*—called by giFT to initiate a query in the P2P network handled by the plugin.
- **download_start*—called by giFT to request the plugin to start a download.
- **share_add*—called by giFT to add a new (local) file to the list of files shared on the P2P network supported by the plugin.
- **share_remove*—called by giFT to remove a file from the list of (local) files shared on the P2P network supported by the plugin.
- **stats*—called by giFT to retrieve relevant statistics from the plugin (e.g. number of search requests, number of download requests, number of files shared, etc.).

4 The Software Architecture of the Turtle Plugin

The software architecture of the Turtle plugin for giFT is shown in Figure 6 (at the end of this document).

As shown in Figure 6 the software architecture of the Turtle plugin consists of the following components:

- The TCP Channel Manager.
- The Router.
- The File Service.
- The Query Service.
- The Turtle Service.
- The giFT integration layer.

In the following sections we will discuss each of these modules.

4.1 The TCP Channel Manager

Turtle nodes are interconnected via TCP/IP connections. Each node maintains a set of neighbors (trusted nodes owned by friends). The set is configured by the user and is not dynamically updated from the network as in Gnutella. To add a new neighbor the user must enter neighbors ID, IP address and secret master key. The same must be done by the other side, otherwise the nodes cannot establish connection.

The TCP Channel Manager is responsible for all communication-related tasks. The *TCPChannelManager* class is defined in *netbase/TCPChannel.h* (in the Turtle source directory), and implemented in *netbase/TCPChannel.h*. The TCP Channel Manager accepts TCP connections coming from neighbors and also periodically tries to contact them. If a TCP connection is established, the authentication procedure begins. During the authentication procedure, two session keys (one for each direction) are generated and securely exchanged using the shared master key. Data streams are then encrypted with session keys. Immediately after authentication a *CommunicationChannel* object is created and registered at Router. The *CommunicationChannel* object is defined in *netbase/CommunicationChannel.h* (in the Turtle source directory), and implemented in *netbase/CommunicationChannel.cpp*.

From that moment it is possible to create virtual circuits through newly created channel. When TCP Channel Manager receives data from a TCP connection, it passes the data to corresponding Communication Channel. The Communication Channel unmarshalls commands from the data and passes them to the Router. The inverse situation looks as follows: when a Communication Channel receives command from the Router, it marshalls the command to a buffer and passes it to the TCP Channel Manager. TCP Channel Manager adds MAC (Message Authentication Code), encrypts everything and sends it to the neighbor node. Because multiple virtual circuits may be tunneled through one TCP connection, data multiplexing is necessary to guarantee fair division of connection bandwidth. This task is up to the Communication Channel. It cyclically sends data of circuits and assigns higher priority to channel commands (connect , flow control , close , etc.). Because network bandwidth is limited, the Communication Channel might not be able to send data fast enough, and this is why it has to control outgoing data flow of virtual circuits. Each circuit is supplied with send buffer of certain size. Flow control messages are sent to Router to ensure that this buffer is not overfilled. Section 4.2.1 gives more details about possible circuit commands and flow control. Figure 2 shows a simplified picture of how the Communication Channel works.

4.1.1 TCP connection initialization

The TCP connection initialization is a four step procedure. The first three steps mutually authenticate the newly connected nodes using a challenge-response protocol. At the end of authentication procedure, nodes share the following information:

- The node ID of neighbor.
- The session key for outgoing data encryption.
- The initialization vector for outgoing data encryption.
- The HMAC key for outgoing data authentication.
- The session key for incoming data decryption.

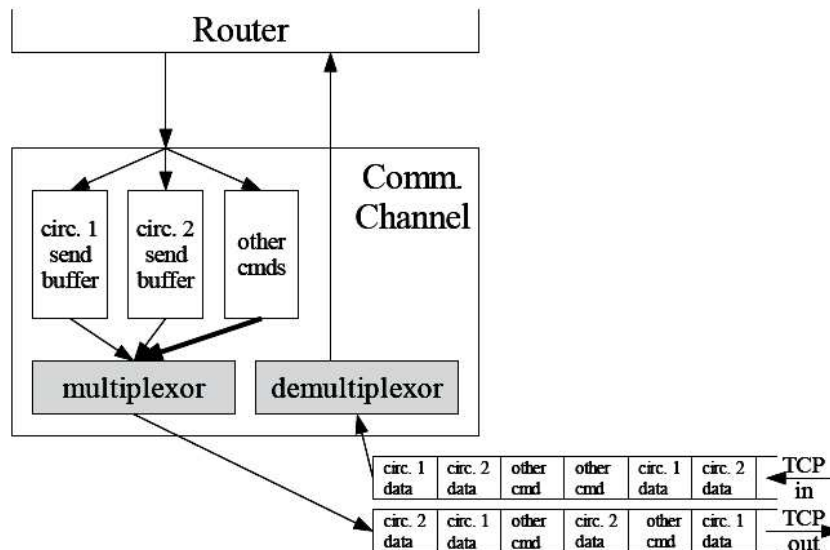


Figure 2. The Communication Channel Module

- The initialization vector for incoming data decryption.
- The HMAC key for incoming data authentication.

The first authentication packet is sent by the initiator of TCP connection (node **A**) to the node that has accepted the connection (node **B**). Table 1 shows its structure:

Length	Description
47 bytes	ID of node A
16 bytes	Random number generated by A

Table 1. The first authentication packet (A → B)

After receiving the first packet node **B** knows identity of node **A**. Node **B** then searches its keystore for the master key shared with this node. If the master key is not found, the connection is immediately closed. Otherwise node **B** sends the second authentication packet (Table 2). The whole packet is encrypted (AES) in CBC mode using the master key shared between **A** and **B**.

Node **A** can now verify node **B**'s identity. It decrypts the second authentication packet with the master key and checks the following conditions:

- The random number at the beginning of the packet must be same as the number in the first authentication packet (to guarantee freshness).
- The ID of node **B** must be same as what is configured at node **A**.

Length	Description
16 bytes	Random number from the first packet.
47 bytes	ID of node B
16 bytes	Random number generated by B
16 bytes	AES session key for data stream A \rightarrow B
16 bytes	AES init vector for data stream A \rightarrow B
16 bytes	HMAC key for data stream A \rightarrow B
20 bytes	HMAC/SHA1 digest of bytes 0-126
13 bytes	Zero padding

Table 2. The second authentication packet (B** \rightarrow **A**)**

- The HMAC digest of the packet must be correct. The digest is computed using the master key.

During the third step node **A** authenticates itself to node **B**. The third authentication packet (Table 3) is again encrypted (AES) in CBC mode using the master key:

Length	Description
16 bytes	Random number from the second packet.
16 bytes	AES session key for data stream B \rightarrow A
16 bytes	AES init vector for data stream B \rightarrow A
16 bytes	HMAC key for data stream B \rightarrow A
20 bytes	HMAC/SHA1 digest of bytes 0-63
12 bytes	Zero padding

Table 3. The third authentication packet (A** \rightarrow **B**)**

Node **B** decrypts the third authentication packet and performs same checks as node **A** in the previous step. After successful authentication both nodes have all information needed to initialize the encryption, decryption, and MAC engine for outgoing and incoming data streams.

Finally, the last step in the initialization procedure activates the connection. The activation is necessary, because neighbor nodes are equal and both may decide to connect at the same time. If this situation occurs, the result is two equal TCP connections, but only one of them may survive. During the activation step the node with the higher node ID decides whether the newly authenticated connection should survive. Because the responsibility for activating the connection lies always on the same node no matter who initiated the connection, it never happens that both connections are activated or that both connections are closed.

4.1.2 Data transfer

Data stream sent via TCP connection is divided into blocks (Table 4). Each block begins with header, which contains length of the data being transferred in that block. The header is followed by the data, the HMAC/SHA1 digest of the data, and padding. Everything but the header is encrypted (AES) in CBC mode. Both AES and HMAC are initialized only once, immediately after authentication.

Length	Description
4 bytes	Data length
1–8192 bytes	Data
20 bytes	HMAC
0–15 bytes	zero padding

Table 4. Structure of a data packet.

4.2 The Router

The router is the module where most of the Turtle “magic” happens. Essentially, this module is responsible with routing data (e.g. queries and query results) hop by hop, from source to destination. The router is designed as a *Router* class, defined in *netbase/Router.h*, and implemented in *netbase/Router.cpp*. Internally, the router holds a list of *router channels* implemented as *RouterChannel* objects (this class is defined in *netbase/Router.h* and implemented in *netbase/Router.cpp*. Each router channel corresponds to either a communication channel (held by the TCP Channel Manager), or to an application channel (held either by the Query Service, or File Service, to be described next). Each router channel contains multiple *router circuits* implemented as *RouterChannelCircuitInfo* structures (also defined in *netbase/Router.h*).

Conceptually, routing consists of passing data from one router circuit to another one. To facilitate this, each *RouterChannelCircuitInfo* structure contains a pointer to the next *RouterChannelCircuitInfo* structure in the router. Additionally, the router also holds an associative map *RCCMap* that maps each *RouterChannelCircuitInfo* structure to the circuit ID of the next circuit in the path (each *RouterChannelCircuitInfo* is uniquely identified by such a circuit ID).

4.2.1 Circuit life-cycle

In this section we describe how circuits are actually established, and how data is sent through them. The circuit lifecycle is managed via commands processed by the router. These commands are stored in a list – *RCmdList* held by the router. These commands are received either from application channels or from TCP channels, as regular data packets on a special command circuit established at initialization time. There are six basic commands related to router circuits:

- *Connect*—is a request to create a new circuit. It is always the first command of the circuit.
- *Connected*—is a reply to the *Connect* command. It is sent when the circuit has been successfully created.
- *Close*—is a request to close the circuit. It may be sent as a reply to connect command, if the circuit cannot be created.
- *Closed*—is a reply to the *Close* command. It confirms that the circuit has been closed. It is always the last command sent to the circuit.
- *Forward*—transfers data through the circuit. It can only be sent on established circuit after the *Connected* command has been received or sent and before the *Close* command is received or sent.

- *Flow Control*—prevents congestion of the data receiver, in the case it cannot process received data fast enough. It can only be sent on established circuit. This command gives certain amount of credits to send more data. Each credit permits the receiver of this command (i.e. data sender) to send one more byte.

The reason for having separate *Close* and *Closed* commands is that it gives us exact the moment when the circuit can be forgotten, because no more circuit commands may arrive. Without the *Closed* command, it would be possible to ignore commands for unknown circuits, but this has two disadvantages. First, circuit IDs could not be reused. Second, it is less immune to errors in implementation.

4.2.2 Creating circuits

In order to establish a virtual circuit, the *Connect* command needs to be provided with a target address. The target address represents the end point of the circuit, which can either be a service on the local Turtle node, or a service on a remote node. The general modus operandi is that when the router receives a *Connect* command from a TCP or application channel it examines the target address, retrieves the information about the next hop, and forwards the *Connect* command to the appropriate neighbor or local service. A circuit with multiple segments is established hop by hop in this way.

When the router receives a *Connect* command, it needs to parse the target address in order to figure out where the command should be forwarded. The target address always consist of the ID of the channel where the circuit continues. However, the router also needs to provide the target address for the next hop in the circuit path. This is accomplished by having each node hold a routing table. This table is held part of a *StatefulAddressManager* object (defined in *netbase/StatefulAddressManager.h*, and implemented in *netbase/StatefulAddressManager.cpp*) part of the *Query Service* (this is the only service that needs to establish new circuits for downloading query results). This routing table is simply a mapping between the query IDs and the channels on which the respective queries came from.

4.2.3 Command routing

We will explain now what happens when circuit commands arrive at a node. These commands are received from either a TCP channel or from an application channel, and passed to the router. The router maintains a table of opened circuits for each channel (the *RCCMap* associative map described earlier). Each circuit has a record in the table saying where it continues (to which channel), and what is the circuit segment ID at the target channel. Based on this information the router decides where to forward the command. It passes the command to selected channel, which undertakes responsibility for the command.

There are two types of channels that can be registered at the router. One of them is the TCP channel. The other one is the application channel, which has the role of end point of virtual circuits. For the router there is no difference between these two channels: both have the channel interface, so they accept and invoke the *Connect*, *Connected*, and so on commands. The router does not see that a TCP channel passes all commands to a neighbor node, while an application channel processes everything locally. The reason to put an application channel component between the router and a service is that it has an easy to use, BSD sockets-like, interface. It makes development of Turtle services easy for programmers with knowledge of BSD sockets library.

4.3 The Query Service

The Query Service implements the Turtle query protocol. Figure 3 shows its internal structure:

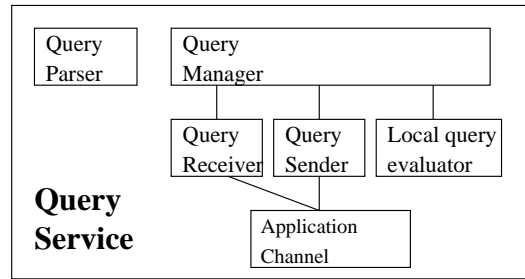


Figure 3. The internal structure of the Turtle Query Service

When a query is first entered by the user, it is parsed by the *Query Parser* on her Turtle node in order to check correct syntax. The query parser is implemented as a *BasicQueryUtility* object which extends the *QueryParser* interface defined in *netsvc/querytypes.h*. The class for the *BasicQueryUtility* object is defined in *netsvc/BasicQueryUtility.h* and implemented in *netsvc/BasicQueryUtility.h*. The query parser processes the query text (as entered by the user) and generates a *query parse tree*. In this tree, leaf nodes are simple text tokens, while intermediate nodes are relational and logical operators. Once the query parse tree is constructed, it is then incorporated into a query packet and flooded into the Turtle network. Although using parse tree instead of original query string might slightly increase size of query packet, this also reduces the CPU load of nodes in the Turtle network, because parsing is done only once and not on every node of the query broadcast tree. Query packet is then inserted into the Turtle network and is propagated until the desired maximum depth.

Queries are sent using the *Query Sender* module. This module is implemented as a *QuerySender* object (the corresponding class is defined in *netsvc/QuerySender.h* and implemented in *netsvc/QuerySender.cpp*). The query sender maintains set of virtual circuits permanently connected to all neighbor nodes. When a new neighbor appears, the query sender contacts neighbor's *Query Receiver* module and establishes a virtual circuit. This circuit is used for sending queries and receiving query hits. For the other direction of communication—receiving queries and sending query hits—there is second circuit initiated by neighbor's query sender, which is connected to our the local query receiver. Having two circuits with simple protocol instead of one with complex protocol makes implementation easier. The *Query Receiver* module is implemented as a *QueryReceiver* object (the corresponding class is defined in *netsvc/QueryReceiver.h* and implemented in *netsvc/QueryReceiver.cpp*).

When the Query Receiver receives a query from a neighbor node, it passes it to the *Query Manager* module. This is implemented as a *QueryManager* object (the corresponding class is defined in *netsvc/QueryManager.h* and implemented in *netsvc/QueryManager.cpp*). The Query Manager is responsible for detecting collisions (the same query reaches a node more than once) caused by cycles in friendship graph. For this purpose, it maintains table of recently seen queries (this is defined as *HashTableWithAge<QueryId, QRecord> table*; in *netsvc/QueryManager.h*). If the received query is found in the table, it means there is a collision, and the query is ignored (since it has been already processed). Otherwise the query is passed to the Query Sender, which forwards it to all neighbors except the one that originated the query. The query is also passed to the *Local Query Evaluator* module,

which evaluates the query against locally shared files.

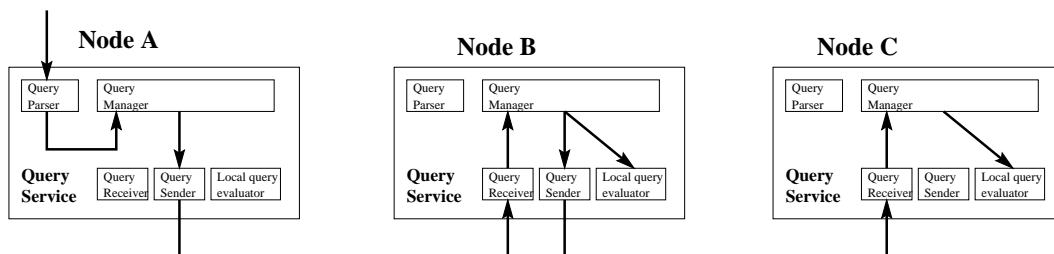


Figure 4. Query propagation through a small Turtle network

The Local Query Evaluator is implemented as a *LocalQueryEvaluator* object (the corresponding class is defined in *netsvc/LocalQueryEvaluator.h* and implemented in *netsvc/LocalQueryEvaluator.cpp*). If the Local Query Evaluator finds a hit, it builds a query hit packet that contains all attributes of the hit (file name, file length, etc.), the address (application channel ID) of the local *File Sender* module, and the query ID of corresponding query. The query hit packet is passed to the Query Manager, which finds the query ID in the table of recently seen queries (*HashTableWithAge<QueryId, QRecord>* table; in *netsvc/QueryManager.h*). The table also contains the query source (again in terms of channel ID) for each query. With this information, the Query Manager passes the query hit packet to the Query Receiver, which sends it to appropriate neighbor via one of the permanently connected virtual circuits.

When the Query Sender receives a query hit packet from a neighbor node, it first updates the destination address in the packet with the address (channel ID) of the TCP channel to the node where the hit came from. The modified query hit packet is then passed to the Query Manager, where it is handled in similar way as query hits received from Local Query Evaluator—it is forwarded either to the Query Receiver or to the local application. Figures 4 and 5 give an example of how query and hit packets are propagated through a small Turtle network.

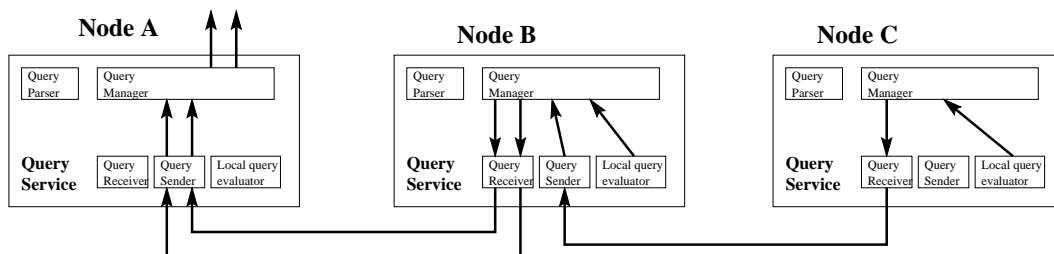


Figure 5. Hit propagation through a small Turtle network

4.3.1 Query syntax

The basic building block of a query is formula with attribute name, relational operator and a value (if the operator is not unary):

- *attribute* > *value*—True if attribute is greater than value.

- $attribute \geq value$ —True if attribute is greater than or equal to value.
- $attribute == value$ —True if attribute is equal to value.
- $attribute \leq value$ —True if attribute is less than or equal to value.
- $attribute < value$ —True if attribute is less than value.
- $attribute = value$ —True if value is a substring of attribute.

Neither attribute name nor value may contain spaces. However if the value is enclosed in quotation marks or single quotes, spaces are allowed. Backslash is used as an escaping character to insert quotation marks or single quotes to the value. String comparisons are case insensitive. Basic formulae are connected with logical operators:

- $formula_1 AND formula_2$ —True if both formulae are true. Ampersand (&) is a shortcut for the *AND* operator.
- $formula_1 OR formula_2$ —True if at least one of formulae is true. Pipe (|) is a shortcut for the *OR* operator.
- $NOT formula$ —True if formula is false. Exclamation mark (!) is a shortcut for the *NOT* operator.

More complex logical expressions can be built using parentheses.

4.4 The File Service

The *File Service* is a service that transfers files over the network. It has two modules—the *File Sender* and the *File Receiver*. The File Service is implemented as a *FileService* object (the corresponding class is defined in *netsvc/FileService.h* and implemented in *netsvc/FileService.cpp*). The File Sender is implemented as a *FileSender* object, while the File Receiver is implemented as a *FileReceiver* object (with the corresponding classes defined in *netsvc/FileSender.h* and *netsvc/FileReceiver.h*, and implemented in *netsvc/FileSender.cpp* and *netsvc/FileReceiver.cpp*).

When a Turtle user decides to download a file from the network, he first uses the Query Service to locate the file. Besides other attributes, the Query Service returns the name of file and the next hop address towards the File Sender of node where the file resides (each node along the path holds the “next next” hop address). The user then asks local the File Receiver module to download the file.

The File Receiver manages all file downloads. When asked to download a file, it opens a virtual circuit to the remote File Sender on the node that has the file, and sends request packet. The packet contains file name, starting position and maximum length of data. The remote File Sender finds the file on its filesystem and responds with the desired data. During the transfer all data received by the File Receiver is stored on local filesystem at location specified by the user. After the transfer completes, the user is notified that the file is ready. The structure of packets used in communication between the File Receiver and the File Sender is shown in Tables 5 and 6. These structures, as well as other important constants (the commands accepted by the File Service for example) are defined in *netsvc/filetypes.h*.

The protocol described here supports continuing of interrupted downloads, which is an important feature, because the chance that a given circuit in the Turtle network will be interrupted is much higher

Length	Description
4 bytes	Command (FSVC_CMD_GET_REQ)
4 bytes	Length of the whole packet
4 bytes	Position where to start the download
4 bytes	Maximum number of bytes to download
1–255 bytes	File name

Table 5. The file request packet

Length	Description
4 bytes	Command (FSVC_CMD_GET_REP)
4 bytes	Length of the whole packet
4 bytes	Error code
4 bytes	Position where the download starts
1–(MAX_INT-1) bytes	File data

Table 6. The file reply packet

than in normal peer-to-peer networks. This happens whenever any node on the path is shut down. Circuits used to download long files are especially vulnerable to this problem. When a download is interrupted, it is usually not possible to restart it by simply connecting to the same address as before, even when the target node is up. The reason is that the address routes the circuit through node that is probably down. That is why a new query must be issued to find different route to the target node. After the node is found (or different node with the same file), the download can be restarted from the place where it was interrupted.

4.5 The Turtle Service

Given these basic building blocks, the Turtle plugin is actually implemented as a *TurtleService* object (the actual class is defined in *netsvc/TurtleService.h* and implemented in *netsvc/TurtleService.cpp*). The *TurtleService* object contains pointers to the objects implementing all relevant modules (e.g. TCP Channel Manager, Router, Query Service, and File Service). These objects are created when the Turtle Service is initialized. The Turtle Service offers a simple interface exposing the functionality expected from a Turtle node. The most important methods are:

- *parseQuery()*—creates a query parse tree from a query string, by passing it to the Query Service (which in turn passes it to the Query Parser module).
- *query()*—initiates a query by passing a query packet (which contains a query parse tree) to the Query Service (which in turn passes it to the Query Manager module—see Section 4.3).
- *addFile()*—adds a new file to the list of shared files. These are the files against which incoming queries are checked.
- *removeFile()*—removes a file from the list of shared files.

- *registerNeighbour()*—registers a new neighbour node. Neighbour nodes are those operated by friends (and the only one to which a Turtle node can connect).
- *deregisterNeighbour()*—removes a node from the neighbour nodes list.
- *connectNeighbour()*—connects to a neighbour node, by passing a *Connect* request to the TCP Channel Manager.
- *disconnectNeighbour()*—tears down the connection to a neighbour node.
- *downloadFile()*—starts downloading a file from another Turtle node. This request is passed to the File Service (which in turn passes it to the File Receiver module).

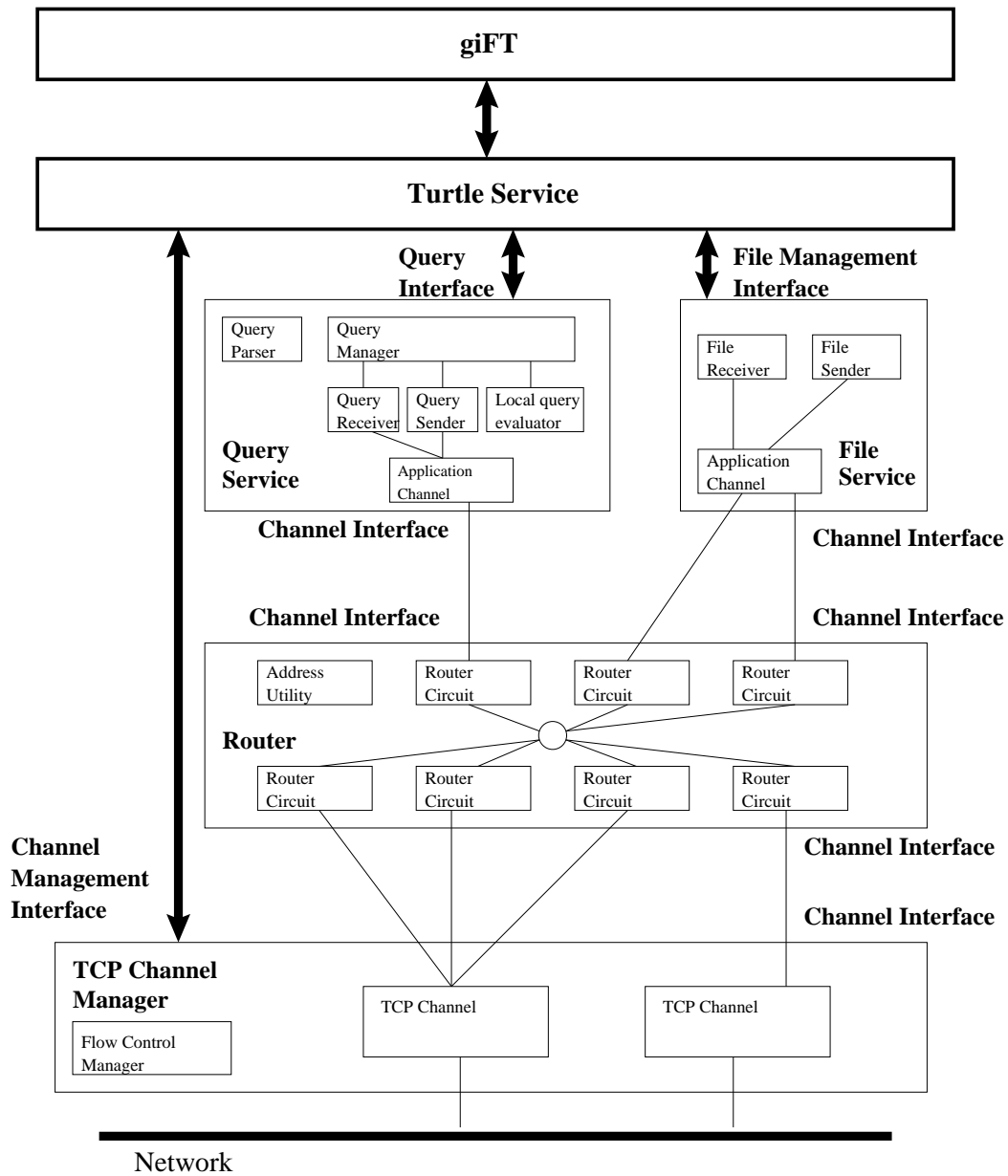


Figure 6. The software structure of the Turtle plugin for giFT