Onion Routing Network Requirements Document

1. Introduction

This document specifies the requirements for each of the sub-systems of the onion routing network, version number 2.0, and for the common data objects shared by those sub-systems.

2. Common Data Objects

This section describes the common data objects, their content, and requirements for operations related to their use.

2.1 Cells

A cell is a data structure consisting of a cell header section and a payload section. The cell is the minimum message unit transmitted on an anonymous circuit or a virtual pipe. Cells carry both user application data and network control and maintenance data.

2.1.1 Cell Structure

Cells consist of a header section and a payload section. Cell header structures are common to all cells. However, cell payload structures are different for each type of cell. The following diagram outlines the cell structure.

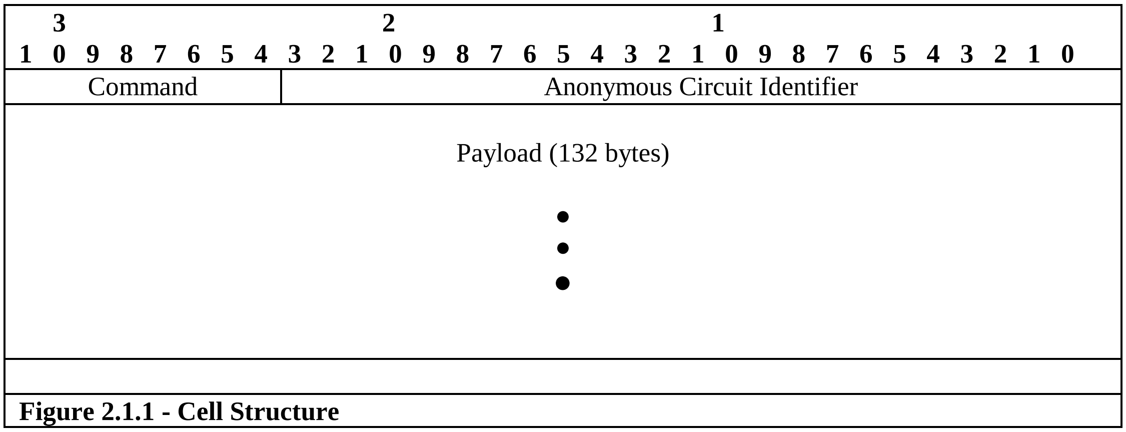

Figure 2.1.1 - Cell Structure

All of the fields in all of the various cell types, except an ONION_INFO cell, are formatted in network byte order. Therefore, conversion routines such as htonl(3N), htons(3N), ntohl(3N), and ntohs(3N) must be used when formatting or retrieving data from cell fields respectively. The ONION_INFO cell is different because the data fields in the payload of an ONION_INFO cell are specified in host byte order. The fields of the cell will be explained in more detail in subsequent sections.

2.1.1.1 Cell Header Field Structure

The header consists of two fields, the command field and the anonymous circuit identifier.

The command field is one byte and identifies the cell type. The command field can be set to one of the following symbolic values:

a) CREATE
b) DATA
c) DESTROY
d) PADDING
e) ONION_INFO
f) DB_UPDATE
g) DB_QUERY

h) ACI_RANGE_ALLOC
    i. ACI_ASSIGN
    ii. ACI_ACK
    iii. ACI_NAK
    iv. ACI_ABORT

The anonymous circuit identifier (ACI) is three bytes and identifies the anonymous circuit upon which the cell is being routed. Cells on an ordinary network are commonly associated with virtual circuits, so cells in the Onion Routing network are similarly associated on a particular virtual pipe with anonymous circuits which have identifiers called anonymous circuit identifiers. The ACI field of a cell indicates the anonymous circuit that the cell is being transmitted on. This ACI is unique on a given virtual pipe but they are not globally unique. Two anonymous circuits on two different virtual pipes within an onion router core can share a given ACI, and further onion router cores know nothing of the ACI's used on other onion router cores. It is possible that a given circuit may be assigned the same ACI along two different hops of the circuit but adjacent hops must use different identifiers for their parts of the circuit.

2.1.1.2 Cell Payload Structure

The payload is 132 bytes long and consists of different informational fields depending on the cell type. Processing of cell payloads is dependent on the cell type also.

2.1.1.2.1 DATA Cell Payload Structure

The DATA cell payload is used to transport user data through the onion network. The DATA cell payload contains network operational data when it is an additional portion of a layered onion as defined in section 2.4.

2.1.1.2.2 DESTROY Cell Payload Structure

The DESTROY cell payload is undefined. The cell size remains the same for this cell type, but the content of the payload is undefined.

2.1.1.2.3 PADDING Cell Payload Structure

The PADDING cell payload is undefined. The cell size remains the same for this cell type, but the content of the payload is undefined.

2.1.1.2.4 CREATE Cell Payload Structure

The CREATE cell payload is a portion of the layered onion as defined in section 1. The onion is apportioned over as many CREATE cells as is necessary to contain the entire onion. An onion layer is 24 bytes. A single onion has 11 layers, so 3 cells are required to contain an entire onion.

2.1.1.2.5 ONION_INFO Cell Payload Structure

The ONION_INFO payload contains the information that is contained in a given layer of an ordinary onion. These information fields are:

Field Name	Length (bytes)
Version	1

Field Name	Length (bytes)
forward crypto function index	1
reverse crypto function index	1
Flags	1
expiration time	4
forward crypto key	20
Forward crypto key initialization vector	20
reverse crypto key	20
reverse crypto key initialization vector	20
next hop global identifier	2
SHA1 digest of first 88 bytes of onion	38

Table 2.1.1.2.5A - ONION_INFO Cell fields

There are currently three flags defined:

a) Tunneled route enable
b) In-band signaling enable
c) Replayable circuit enable

The structure of an ONION_INFO cell is as follows.

<table>
<tr><td></td><td>3</td><td colspan="8"></td><td>2</td><td colspan="8"></td><td>1</td><td colspan="11"></td></tr>
<tr><td>1</td><td>0</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">Version</td><td colspan="8">Fwd Crypto Fn Index</td><td colspan="8">Rev Crypto Fn Index</td><td colspan="8">Flags</td></tr>
<tr><td colspan="32">Expiration Time (4 bytes)</td></tr>
<tr><td colspan="32">Forward Crypto Key (20 bytes)
●
●
●</td></tr>
<tr><td colspan="32">Forward Crypto Key Initialization Vector (20 bytes)
●
●
●</td></tr>
<tr><td colspan="32">Reverse Crypto Key (20 bytes)
●
●
●</td></tr>
<tr><td colspan="32">Reverse Crypto Key Initialization Vector which is the same as the Forward Crypto Key Initialization Vector except the first 10 bytes and last 10 bytes are switched (20 bytes)
●
●
●</td></tr>
<tr><td colspan="16">Next Hop Global Identifier (2 bytes)</td><td colspan="16">First two byes of SHA1 Digest (2 bytes)</td></tr>
<tr><td colspan="32">SHA1 Digest of the first 88 bytes above (36 bytes)
●
●
●</td></tr>
<tr><td colspan="32"></td></tr>
</table>

Table 2.1.1.2.5B - Structure of ONION_INFO cell

As the ONION_INFO cell is meant to be easily interpreted by the onion router core all of the data fields in the cell will be formatted in the host byte order and thus it is not required to use conversion routines such as ntohl(3N) and ntohs(3N) to properly access the data.

2.1.1.2.6 DB_QUERY Cell Payload Structure

The DB_QUERY cell payload contains the following fields.

Field Name	Length (bytes)
query type	2
GID of Query Initiator	2
GID of Query Initiator Peer	2

Table 2.1.1.2.6A - DB_QUERY Cell Payload Fields

The query type field identifies the type of query being made. The valid query types are the following.

Query Type	Value of query type filed	Description
local topology	DB_QUERY_TYPE_TOPO_LOCAL	request from a DBE to its associated sub-system for link status of all that sub-systems virtual pipes
global topology	DB_QUERY_TYPE_TOPO_GLOBAL	request from a DBE to a second DBE for a complete copy of the second DBE’s topology graph
specific topology	DB_QUERY_TYPE_TOPO_SPECIFIC	request from a DBE to a second DBE for link status of the specific virtual pipe between id1 and id2
local key	DB_QUERY_TYPE_KEY_ LOCAL	request from a DBE to a onion router core for its public key certificate
global key	DB_QUERY_TYPE_KEY_ GLOBAL	request from a DBE to another DBE for a complete copy of all public key certificates held by the second DBE.
specific key	DB_QUERY_TYPE_KEY_ SPECIFIC	request from one DBE to a second DBE for a copy of the public key certificate of the onion router core specified in id1

Table 2.1.1.2.6B - DB_QUERY Cell Types

2.1.1.2.7 DB_UPDATE Cell Payload Structure

The DB_UPDATE cell payload contains the following fields.

Field Name	Length (bytes)
update type	2
number of updates in cell	2
update data	variable

Table 2.1.1.2.7A - DB_UPDATE Cell Payload Fields

The content of the update data field is dependent upon the value of the update type field. There are currently two types of update defined.

Update Type	Value of update type filed	Description
Topology	DB_UPDATE_TYPE_TOPO	Link state updates
Key	DB_UPDATE_TYPE_KEY	Public key updates

Table 2.1.1.2.7B - DB_UPDATE Cell Types

2.1.1.2.7.1 DB_UPDATE Cell Payload Structure– Topology Update Data

The update data field in a DB_UPDATE cell of the topology update type consists of up to two records containing the following fields.

Field Name	Length (bytes)	Description
Initiator GID	2	global id of initiator of update
Peer GID	2	global id of initiator peer
time	4	time of update by initiator’s clock
status	2	link status: Up Down Unknown
signature	38	SHA1 digital signature of update record as signed by id1
Table 2.1.2.2.5.1 - Topology Update Record Format		

Each topology update record is 48 bytes long. Thus in the 132 byte payload of a DB_UPDATE cell at most two updates can be included. All remaining payload bytes, either 84, in the case of a single update, or 36, in the case of two, will be set to random values.

2.1.1.2.7.2 DB_UPDATE Cell Payload Structure - Public Key Update Data

A public key update record spans multiple DB_UPDATE cells due to the length of the actual key certificate. Each of the cell payloads consists of the following fields.

Field Name	Length (bytes)	Description
GID	2	owner of public key certificate
index	2	sequence number of update this cell contains
certificate	variable	actual certificate data
Table 2.1.1.2.7.2 - Public Key Update Fields		

2.1.1.2.8 ACI_RANGE_ALLOC Cell Payload Structure

ACI refers to an Anonymous Connection Identifier. The ACI_RANGE_ALLOC cell is used to assign the ACI range that will be used on a given virtual pipe. The information fields contained in the ACI_RANGE_ALLOC cell payload are as follows.

Field Name	Length (bytes)
ACI lower bound	4
ACI upper bound	4
Command (ASSIGN, ACK, NAK, ABORT)	2
Table 2.1.1.2.8A - ACI_RANGE_ALLOC Fields	

As ACI's currently only use 3 bytes the most significant byte will be set to zero.

The command values, ASSIGN, ACK, NAK, and ABORT are used to distinguish the meaning of the ACI_RANGE_ALLOC cell during the ACI range negotiation protocol. This protocol is discussed in section 2.3.7.3.

The structure of the ACI_RANGE_ALLOC cell is as follows.

3 1 0 9 8 7 6 5 4	2 3 2 1 0 9 8 7 6	1 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0	ACI lower bound	
0	ACI upper bound	
0		Command

Table 2.1.1.2.8B - ACI_RANGE_ALLOC Cell Structure

2.1.2 Circuit vs. Pipe Oriented Cells

Of the various cell types some are circuit oriented, i.e. they carry information related to a specific anonymous circuit, and some types are virtual pipe oriented, i.e. they do not refer to any particular circuit. The circuit oriented cell types are:

a) CREATE
b) DATA
c) DESTROY
d) ONION_INFO

The pipe oriented cell types are:

a) PADDING
b) DB_UPDATE
c) DB_QUERY
d) ACI_RANGE_ALLOC
    i. ACI_ASSIGN
    ii. ACI_ACK
    iii. ACI_NAK
    iv. ACI_ABORT

The difference in processing these cell types is that circuit oriented cell type processing is dependent upon the state of the anonymous circuit and pipe oriented cell processing is not.

2.2 Anonymous Circuits

Anonymous circuits are used to carry user and control data through the onion network. Each application connection request maps to a unique anonymous circuit. Anonymous circuits carry cells from an application proxy to an output funnel or reply onion proxy.

2.2.1 Definitions

A anonymous circuit is a path through the onion network that carries cells from a source to a destination. An anonymous circuit consists of an application proxy at one end and an responder proxy or reply onion processor at the other. In between these two endpoints are some number of onion router network sub-systems that forward the cells along the route.

An anonymous circuit master is the network element that holds the symmetric encryption keys for the circuit. For ordinary circuits this is the application proxy but for reply circuits it is the reply onion processor.

An anonymous circuit participant is any network element through which a circuit is routed other than the circuit master.

A connection initiator is the application or proxy for which the connection is established.

A connection acceptor is the application or proxy that will accept a connection request.

The above two terms can refer either to the actual application or to the proxy that acts as the interface between that application and the onion network. It should be clear from context which is meant.

2.2.2 Anonymous Circuit Encryption

Anonymous circuits are encrypted at every hop using symmetric cryptography. A separate key is used for each direction on the circuit. Onion router cores simply add their layer of encryption as the circuit cells are routed through them. It is the responsibility of the application proxy to precrypt data destined for the output funnel so that it appears as plaintext when it arrives there, and to postcrypt data destined for the application to obtain the plaintext for the application.

Keys for anonymous circuit encryption are obtained from the onion that created the circuit. Within each onion is 16 bytes of key seed material that is hashed three times to obtain three separate keys and an initialization vector. The last two keys are used for the forward and reverse circuits respectively, the first key is used to complete the decryption of the received onion. The Secure Hash Algorithm (SHA) is used to perform the hashing.

2.2.3 Lifetime of Anonymous Circuits

Anonymous circuits are created and destroyed. In general an anonymous circuit is created in response to an application connection request. When the application is done using the connection the anonymous circuit is destroyed. Anonymous circuits can be destroyed before the application is finished with them if a network error occurs during their use. It is the responsibility of the application proxy to convert a network error code into a meaningful, application specific error, and forward this error back to the application.

2.2.4 Anonymous Circuit States

The following table describes the five possible states of an anonymous circuit. How the onion router core processes certain cell types depends upon the state of the anonymous circuit that the cell is received on.

Anonymous Circuit State	**Description**
Down	An implicit state indicating that no data structure have been created for this circuit. The circuit does not yet exist.
Created	The state that the circuit is in after the receipt of the first CREATE cell for that circuit and up to the time that the last CREATE cell is forwarded to the next hop in the circuit or to the output funnel if this core is the last hop of a circuit. A circuit being in the created state means that a Cryptographic Processor is either processing an onion used for creating the circuit, or that it is awaiting the receipt of the rest of the cells comprising that onion. A circuit in the created state is not yet ready to forward data along itself. It will, however, buffer any DATA cells it receives for itself and will forward these DATA cells immediately upon entering the up state, in the case of non-replayable circuits, or the awaiting_rekey state in replayable reply circuits.

Anonymous Circuit State	Description
Awaiting_Rekey	A replayable reply circuit enters this state after leaving the created state. The circuit in this state is waiting for the DATA cell containing a rekey onion that will rekey the circuit to make it safe for replayability. After receiving and processing the rekey onion, and rekeying the circuit encryption engines with the new keys the circuit will leave the awaiting_rekey state and enter the up state. Note that only a replayable reply circuit ever enters the awaiting_rekey state.
Up	A circuit enters the up state immediately after the last CREATE cell for this circuit is forwarded to the next hop in the circuit. A circuit in the up state is ready to forward data along itself.
Pending	A circuit enters the pending state on an onion router core in one of two ways: Either the core receives a DESTROY cell from a peer sub-system; or the core detects a virtual pipe error requiring that the circuit be terminated and sends a DESTROY cell down the opposite side of the circuit as the pipe break. A circuit in the pending state does not process any cells except for the destroy acknowledgment which immediately removes the circuit from use.
Active_Kill	A circuit enters the Active_Kill state when a circuit participant sends DESTROY cells down both sides of the anonymous circuit. This is different from the pending state in that the circuit participant has no knowledge of the state of its peer participants. In the pending state the participant knows that one of the participant peers has taken down the circuit.

Table 2.2.4 - Anonymous circuit States

2.2.5 Anonymous Circuit Types

There are six different types of anonymous circuits that can be created. These circuit types are:

a) Ordinary circuits
b) Reply circuits
c) Replayable reply circuits
d) Tunneled circuits
e) Tunneled reply circuits
f) Tunneled replayable reply circuits

2.2.5.1 Ordinary circuits

Ordinary circuits are the circuits that are expected to be used most of the time. In these circuits the connection initiator creates the onion and acts as the master of the circuit. The route of the circuit has few enough hops so that a single onion can be used to create the circuit.

2.2.5.2 Reply Circuits

Reply circuits are those where the connection acceptor has created the onion and acts as the circuit master. In such a circuit the connection initiator has obtained a reply onion from some repository and is using it to make a connection to an anonymous connection acceptor. The route is short enough so that a single onion can be used to create the circuit.

2.2.5.3 Replayable reply circuits

Replayable reply circuits are reply circuits in which it is permissible to use the given reply onion multiple times. This is forbidden in non-replayable circuits.

2.2.5.4 Tunneled circuits

Tunneled circuits are those for which the circuit route has too many hops to fit into a single onion and thus multiple onions are needed to create the circuit.

2.2.5.5 Tunneled reply circuits

Tunneled reply circuits are reply circuits that have too many hops to fit into a single reply onion. Thus the overall reply onion package actually contains two or more onion structures as well as the additional connection information.

2.2.5.6 Tunneled replayable reply circuits

Tunneled replayable reply circuits are replayable reply circuits that have too many hops to fit in a single onion. Thus the overall reply onion package actually contains two or more onion structures as well as the additional connection information.

2.2.6 Circuit Master to Core Covert Signaling

Application proxies have the capability to send covert signals to a given onion router core along an anonymous circuit. This capability will not be used in this version but the capability for covert signaling must be in place in this version.

The signaling is covert because only the circuit master and the core being signaled know that such a signal is being sent. Other circuit participants remain unaware of this fact.

The method for covert signaling involves sending a signaling tag that is unique to each onion router core in the payload of a DATA cell. This tag is chosen to be long enough so that the probability of an actual data message containing this tag is acceptably remote.

The currently anticipated tag length is half the payload of a DATA cell or 66 bytes. Assuming independent bits each with equal probabilities of being a one or a zero we have a data/signal tag collision probability of (0.5)^(66*8) = 2^(-528) = 1.14E(-159). Thus it is much more likely to get undetectable TCP bit errors than it is for the signal tag and actual data to coincide.

The second half of the payload is used to carry information associated with the signal.

It is the responsibility of the onion router core to check the contents of each DATA cell payload for this signal tag and act on it accordingly. DATA cells containing a signal are not forwarded along the circuit.

It is possible to create a circuit that will not check for signals. This is done by setting a bit in the flag field of the onion creating the circuit that indicates that signal checking will not be performed. In this version of the onion network software having this bit set in the onion flags field will be the default thus relieving the onion router cores of having to perform the relatively laborious check of the payload of all DATA cells. Future versions will make use of this signaling mechanism.

2.3 Virtual Pipes

Virtual pipes are used to inter-connect the sub-systems in the onion routing network. All communications within the onion routing network is performed via virtual pipes with the following two exceptions:

a) the connection initiator application to application proxy connection is as dictated by the application. Currently this is always a stream socket connection.

b) the output funnel to connection acceptor application is as dictated by the application. Currently this is always a stream socket connection.

2.3.1 Virtual Pipe Definition

A virtual pipe is defined as a IPC stream socket connection that transmits and receives cells. A cell is the minimum quanta of information transfer that can take place on a virtual pipe.

2.3.2 Virtual Pipe Link Encryption

Virtual pipes are optionally link encrypted using a symmetric encryption algorithm. The currently used algorithm is DES using output feedback mode. The encryption algorithm specified in the onion router core's configuration file for each virtual pipe used by that core. Additionally a null encryption algorithm can be specified that would have the effect of not performing any link encryption.

Link encryption of virtual pipes is optional. A flag will indicate whether link encryption is to be performed on a given pipe. Note that encryption with the NULL encryption algorithm is considered link encryption, i.e. the flag would indicate that link encryption is enabled. A sub-systems configuration file will specify, for each virtual pipe that sub-system maintains, the link encryption status for that pipe. Peer sub-system administrators must ensure that their respective configuration files are in agreement as to whether or not link encryption will be performed on the virtual pipe connecting those peers.

2.3.3 Virtual Pipe Connection Types

Virtual pipes use an underlying transport mechanism for sending and receiving cells. Currently four transport types are supported and more may be added in future versions.

2.3.3.1 TCP Socket

Virtual pipes between sub-systems running on separate machines will generally use the Berkeley TCP socket as the underlying transport. The domain is AF_INET, and the type is SOCK_STREAM. The sockets will be specified to be non-blocking, but asynchronous i/o will not be used. The following socket options will be set as described in the table below:

Socket Option	Setting
SO_BROADCAST	Off
SO_DEBUG	Off (maybe on during debug phase)
SO_DONTROUTE	???
SO_KEEPALIVE	On
SO_LINGER	On, 10 second linger
SO_RCVBUF	Adjustable per pipe
SO_RCVLOWAT	Adjustable per pipe - not used by O/S
SO_RCVTIMEO	Adjustable per pipe - not used by O/S
SO_REUSEADDR	On
SO_SNDBUF	Adjustable per pipe
SO_SNDLOWAT	Adjustable per pipe - not used by O/S
SO_SNDTIMEO	Adjustable per pipe - not used by O/S

Table 2.3.3.1 - TCP Socket Option Settings

2.3.3.2 Unix Pipe

Peer sub-systems that run on the same machine will usually use a Unix stream pipe as a transport. Pipes come in two flavors, named and unnamed. Unnamed stream pipes are usually referred to as just "stream pipes".

Unnamed stream pipes can only be used between processes if there is a parent/child relationship between those processes. Thus for sub-systems that are spawned by another sub-system, such as the responder proxy, which is spawned by the output funnel, unnamed stream pipes will be used. For peer sub-systems without a parent/child relationship a named streampipe will be used.

2.3.3.3 IPSec Socket

An IPSec socket is a TCP socket that conforms to the Internet Protocol Security standards and thus has built in encryption and authentication at the network layer. This negates the need for link encryption but it also requires that the socket options be specified in a particular way. The table below lists the socket options that must be specifically set and lists the values to which they must be set.

Socket Option	Setting

Table 2.3.3.3 - Socket Option Setting on an IPSec Virtual Pipe

2.3.3.4 Possible Future Types of Virtual Pipe Connections

Possible transport mechanism that may be used in future versions of the onion routing network software include System V IPC mechanisms such as shared memory and message queues. Shared memory has the potential to be an extremely fast IPC mechanism at the cost of requiring the use of semaphores to coordinate shared access.

2.3.4 Virtual Pipe Link Encryption Key Exchange

The exchange keys necessary to perform link encryption between the two ends of the pipe is done at the pipe initialization time using some agreed upon key exchange protocol. Some possible examples include the Station to Station protocol, which is the one currently used, the Oakley key determination protocol, or none if the link will be crypted with a null encryption algorithm.

2.3.5 Virtual Pipe Anonymous Circuit Identifier Ranges

Each virtual pipe has a range of ACI's that it may use for numbering the circuits it carries. This range is a subset of the overall possible ACI address space which is 24 bits wide. ACI ranges will always span a power of two number of ACI's. The ACI range for a given virtual pipe is determined at pipe initialization time and will be assigned by the network element that is considered the master of the pipe connection.

The negotiation of ACI ranges is optional. A flag will indicate whether the concept of ACI range has applicability to a given virtual pipe. A sub-systems configuration file will specify, for each virtual pipe maintained by that sub-system, whether a ACI range will apply to that pipe, and thus whether the ACI range negotiation need be performed at pipe initialization. Peer sub-system administrators much ensure that their respective configuration files are in agreement as to whether or not ACI range negotiation should be performed for the virtual pipe connecting those peers.

2.3.6 Virtual Pipe Database Circuit

On each virtual pipe there is a single circuit reserved for routing Database cells, either the DB_QUERY or the DB_UPDATE cell types. This circuit will be labeled with the first ACI in the virtual pipe's allocated ACI range. Each of the network elements involved in sending or receiving Database cells will maintain this Database circuit in it's virtual pipe anonymous circuit database.

Not all onion network sub-systems have an associated Database Processor. For those sub-systems there will be no Database circuit.

On virtual pipes that do not have the concept of ACI range the Database circuit will be assigned an ACI of zero.

2.3.7 Virtual Pipe States

There are only two virtual pipe states up and down. A virtual pipe in the up state has made a socket connection to its peer and has successfully performed the key exchange and keyed it's link encryption engine. A virtual pipe not in the up state is in the down state. A virtual pipe in the down state is not available to send or receive cells.

2.3.8 Virtual Pipe Initialization

Prior to using a virtual pipe it must be initialized. Because pipe initialization can include a cryptographic key exchange protocol which is computationally expensive and having the potential to disrupt timely traffic flow on the other virtual pipes of the sub-system, a child process will be spawned to initialize each virtual pipe used by the sub-system. Because there is the potential for the key exchange protocol or the ACI range allocation protocol to never terminate the sub-system will set a timer when spawning each pipe initialization child process and kill that child if the timer expires before it terminates. If the initialization process is killed prior to termination then the pipe initialization fails and is retried until some maximum number of retries at which point the sub-system continues without using that pipe. If the pipe that cannot be initialized is critical to the operation of the sub-system then that failure will be considered fatal to the sub-system.

There are three steps to initializing a virtual pipe, these are:

a) establishing the connection
b) optionally performing the key exchange protocol to obtain the keys for the pipe link encryption
c) optionally performing the ACI range allocation negotiation protocol to assign the range of valid ACI's to be used on the pipe.

Steps (b) above is only performed if the pipe is configured to use link encryption. Step (c) above is only performed if the the pipe is to have a ACI subrange allocated to it.

Virtual pipe initialization can take place at two times. The first is during the initialization of the network sub-system. At that time the sub-system attempts to initialize all of the virtual pipes appearing in its configuration file. The second time is after a virtual pipe failure. There is a small difference in how the connection is established in the two cases. Connection establishment during virtual pipe recovery is discussed in section ?????

2.3.8.1 Virtual Pipe Connection Establishment

During the initialization of a network sub-system virtual pipe connections are established in the following manner:

a) Those network sub-systems that are designated as master of a particular virtual pipe connection will actively connect to the far end peer by using the standard active connection mechanism i.e.
    a. create a stream socket
    b. bind that socket to the sub-system's address

    c. connect to the far end peer's well known socket

b) Those network sub-systems that are designated as slave of a particular virtual pipe connection will passively connect to the far end peer by using the standard passive connection mechanism (i.e. letting the far end peer actively connect)
    a. create a stream socket
    b. bind that socket to the sub-systems well known address
    c. listen on that socket for incoming connection requests
    d. accept those incoming connection requests

If at any point in the above sequences a failure occurs the sequence will be aborted and retried. An exponential backoff policy will be used to schedule retries. After a specified number of unsuccessful retries the initialization of the virtual pipe will be aborted and an error reported and logged to the sub-system. Constants specifying the exponential backoff parameters and the number of retries that will be attempted will be in the sub-systems configuration file and will be specified for each virtual pipe. The exponential backoff parameters are

a) Initial delay time
b) Exponential factor b

If the first attempt to initialize the virtual pipe fails then the sub-system will wait the initial delay time before retrying the initialization. Subsequent delay times are given by

Delay Time_i = (Initial Delay Time) * $2^{b*(i-1)}$
where i is the attempt number.

2.3.8.2 Virtual Pipe Link Encryption Key Exchange Protocol

The key exchange protocol is used by the two peers of a virtual pipe connection to agree upon the keys that will be used for the pipe link encryption. This protocol is always initiated by the master of the pipe. The specific protocol that will be used for key agreement is specified in each sub-systems configuration file. These configuration files are maintained by the sub-system administrators and thus the administrators of network neighbors must agree to the key exchange protocol offline.

The current vocabulary of key exchange protocols consists of the, the station to station protocol (STS), the OAKLEY key exchange protocol, and the NULL protocol. The NULL protocol is used when the specified link encryption algorithm is the NULL algorithm.

If the virtual pipe is specified not to link encrypt then the key exchange protocol will not be performed.

If the virtual pipe uses IPSec as the underlying transport then the sub-system will rely on the IPSec link encryption and thus will not perform any application level key exchange. The key exchange that will occur is part of the overall operation of the IPSec protocols and thus transparent to the onion routing application.

2.3.8.3 Virtual Pipe ACI Range Allocation Protocol

The ACI range allocation protocol allows the master of a virtual pipe to interactively assign a range of ACI's to be used for circuits traveling on the pipe. The protocol is interactive to allow the slave to notify the master of range conflicts. This is necessary because some network sub-systems require unique ACI ranges for each pipe.

The protocol proceeds as follows.

a) The master selects an ACI range for the slave and formats and sends it a ACI_RANGE_ALLOC cell with the selected range. The cell has the command field set to ASSIGN.
b) The slave receives the cell and checks that the specified range doesn’t conflict with other ACI ranges on other virtual pipes it maintains.

c) If the assigned range does not conflict with other ranges then the slave sends the same cell back to the slave with the command field set to ACK. Else it sends a ACI_RANGE_ALLOC cell back to the master with a suggested ACI range and the command field set to NAK.
d) The master receives the slaves response. If it is an ACK the master sets the virtual pipe ACI range to that just assigned. If it is a NAK the master selects a different ACI range, perhaps the one suggested by the slave and restarts at step (a). If the number of responses received from the slave is greater than a specified number, i.e. the negotiation has taken too many steps without agreement, then the master sends the slave a ACI_RANGE_ALLOC cell with the command field set to ABORT.
e) When the slave receives a ACI_RANGE_ALLOC cell with the command field set to ABORT, it will send the cell back to the MASTER, abort the ACI range allocation protocol, and report the failure to the sub-system.
f) When the master receives a ACI_RANGE_ALLOC cell with the command field set to ABORT it aborts the ACI range allocation protocol and reports the failure to the sub-system.

If at any point in the protocol it is determined that the virtual pipe connection has broken then the virtual pipe recovery procedure will be entered. The protocol still has the potential of failing if the slave sub-system fails in a way that doesn’t actually bring down the virtual pipe. This is handled by having the sub-system set a pipe initialization timer before spawning the child process that will initialize the pipe. If the child process has not successfully initialized the pipe and passed the sub-system back the valid socket descriptor by the time that the timer expires then the child process will be terminated and the pipe initialization will fail.

It is generally intended that the ACI range allocation protocol be performed only at pipe initialization time but it may be invoked by the master of a virtual pipe connection at any time provided there are no active circuits on that pipe. An active circuit is any circuit not in the down state.

If the virtual pipe is specified not to have a ACI range allocation then the range allocation protocol will not be performed.

2.3.9 Virtual Pipe Errors and Error Recovery

Virtual pipes are subject to error conditions. Pipe errors consist of one of the following types of errors:

a) pipe initialization failure
b) pipe select failure (select system call)
c) pipe read failure
d) pipe write failure
e) pipe encryption failure

2.3.9.1 Virtual Pipe Initialization Failure

Pipe initialization failure can occur for the following reasons:
a) Socket allocation error
b) Key exchange protocol failure
c) ACI range allocation failure

2.3.9.2 Virtual Pipe Select Failure

A virtual pipe select failure occurs when an attempt to "select" a virtual pipe fails as indicated by a return code of -1. Selecting a virtual pipe involves calling the system function select(3c) and passing the socket descriptor of that virtual pipe. Selecting a pipe is done to see if there is currently data ready to read on the pipe and is done so that we don't have to block on the call to read(2) if there is no current data pending on the pipe. Select(3c) is also called to determine whether there are pending connections on a listener socket. This is done so that we don't have to block on the accept(3n) call if there are no pending connections on the socket.

There are several error conditions which can cause a select() to fail. The table below lists these conditions along with the action that will be taken should that error condition arise.

Select Error Return Code	Description	Action Taken
EBADF	bad descriptor	if on a virtual pipe attempt pipe recovery if on a listener socket, try to re-open the socket, re-bind it, and re-listen to it.
EINTR	interrupted by signal	retry the select
EINVAL	bad timeout value	system hosed/fatal

Table 2.3.9.2 - Select Error Conditions and Associated Recovery Action

2.3.9.3 Virtual Pipe Read Failure

A virtual pipe read failure occurs when an attempted read of the socket comprising the virtual pipe fails as determined by the return code of the call to read() being equal to -1. There are a variety of error conditions which can cause a read to fail. These are listed below with the action that will be taken should that error condition arise.

Error	Description	Action Taken
EAGAIN	Temporary failure	Retry the read
EBADF	Bad descriptor	attempt pipe recovery
EBASDMSG	N/A	system hosed/fatal
EDEADLK	N/A	system hosed/fatal
EFAULT	bad buffer address	core hosed/fatal
EINTR	interrupted by signal	Retry the read
EINVAL	N/A	system hosed/fatal
EIO	physical i/o error	attempt pipe recovery
EISDIR	N/A	system hosed/fatal
ENOLCK	full lock table	system hosed/fatal
ENOLINK	downed link	attempt pipe recovery
ENXIO	N/A	system hosed/fatal

Table 2.3.9.3 - Select Error Conditions and Associated Recovery Action

The errors for which the action is fatal should never happen with the type of reads that we are doing, i.e. on Berkeley sockets. A error code of that value shows that the system is corrupted or some other catastrophic failure for which we really cannot hope to recover from short of restarting the core and possibly rebooting the host machine.

2.3.9.4 Virtual Pipe Write Failure

A virtual pipe write failure occurs when an attempted write of the socket comprising the virtual pipe fails as determined by the return code of the call to write() being equal to -1. There are a variety of error conditions which can cause a write to fail. These are listed below with the action that will be taken should that error condition arise.

Error	Description	Action Taken
EAGAIN	Temporary failure	Retry the write
EBADF	Bad descriptor	attempt pipe recovery
EDEADLK	N/A	system hosed/fatal
EDQUOT	N/A	system hosed/fatal
EFAULT	bad buffer address	core hosed/fatal
EFBIG	N/A	system hosed/fatal
EINTR	interrupted by signal	Retry the write
EINVAL	N/A	system hosed/fatal
EIO	N/A	system hosed/fatal
ENOLCK	full lock table	system hosed/fatal

Error	Description	Action Taken
ENOLINK	downed link	attempt pipe recovery
ENOSPC	N/A	system hosed/fatal
ENOSR	N/A	system hosed/fatal
ENXIO	N/A	system hosed/fatal
EPIPE	disconnected pipe	attempt pipe recovery
ERANGE	N/A	system hosed/fatal

Table 2.3.9.4 - Select Error Conditions and Associated Recovery Action

2.3.9.5 Virtual Pipe Recovery

Virtual pipe recovery consists of the following steps:

a) taking down the pipe in an orderly fashion
b) bringing the pipe back up

2.3.9.5.1 Bringing Down a Virtual Pipe

A virtual pipe is brought down in an orderly fashion by doing the following:

a) A DB_UPDATE cell is sent to the database engine connected to the sub-system taking down the pipe. The DB_UPDATE cell will indicate that the pipe is down.
b) DESTROY cells are sent along each anonymous circuit on the pipe in the direction opposite the pipe break. Each circuit will enter the pending state.
c) When all DESTROY acknowledgements have been received and each circuit data structure cleaned up the virtual pipe data structure will be cleaned up, all cells currently being held on the pipe will be freed and the associated queues or buffers marked empty. The pipe state will be marked as down. All encryption engine finalization will be done. All other dynamically allocated memory will be freed.

2.3.9.5.2 Bringing Up a Virtual Pipe

A virtual pipe is brought up by doing the following:

a) Establishing the connection
b) Performing pipe initialization

The connection is established as follows:

a) The network sub-system detecting the pipe error will attempt to actively connect to the far end peer's well known listener socket.
b) When the connection attempt is detected at the listener socket one of two possible actions will occur.
    a. If the sub-system making the active connection is the master of the virtual pipe, then the connection will be accepted and the pipe initialization will be initiated. If the slave already had a virtual pipe connection to that master then that old connection will be taken down.
    b. If the sub-system making the active connection is the slave of the virtual pipe, then the connection will be refused, the master will check to see that it has not already established a new connection to this slave and if not will actively connect to that slave. If it did have a connection then that connection is left in place
c) If that connection attempt is successful the peer making the active connection will check it's list of virtual pipes to ensure that the connection just made is not redundant. If it is redundant then the connection that was actively made from slave to master will be closed and the other connection retained.
d) Likewise when a connection is passively accepted a check will be made to ensure that the connection is not redundant. If it is the connection made actively from slave to master will be closed.
e) If the connection attempt is not successful then exponential backoff will be used to schedule the next attempt at connecting to the far end peer.
f) After the maximum number of retries for actively connecting to the far end peer no further active connection attempts will be made. The listener socket will remain open so that the far end peer can attempt an active connection when it comes back up.

g) The numbers for initial retry delay, exponential time constant, maximum retries, etc. will be obtained from that network element's configuration file. This file is maintained by the network element's administrator.

2.3.10 Virtual Pipe Normal Operation

2.4 Onions

Onions are data structures used to create anonymous circuits and to distribute seed material from which the symmetric keys are generated. An onion refers to a specific data object that consists of N layers, each layer of which is separately encoded for a specific onion router core by using that cores public key for encryption.

Each layer of the onion contains the following fields:

Field Name	Length (bytes)	Description
Zero	1 bits	A zero to ensure that RSA works correctly
Flags	3 bits	Control flags indicating circuit options, but the tunnel bit is used only on the last layer
Version	4 bits	Software version, 1 bit major, 3 bits minor
Back Function	4 bits	The cryptographic function for the reverse circuit
Forward Function	4 bits	The cryptographic function for the forward circuit
Next Hop GID	2	The global identifier of the next hop in the circuit
Expiration Time	4	The expiration time GMT seconds since 1/1/1970
Key Seed Material	16	The key seed material from a random number generator used for creating the symmetric keys used for the anonymous circuit encryption

Table2.4A - Onion Data Fields

These fields are arranged in the onion as in the table below.

<table>
<tr><th></th><th>3</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th>2</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th>1</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th></tr>
<tr><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td colspan="3">Flags</td><td colspan="4">Version</td><td colspan="4">Rev Fn</td><td colspan="4">Fwd Fn</td><td colspan="16">Next Hop GID</td></tr>
<tr><td colspan="32">Expiration Time (4 bytes)</td></tr>
<tr><td colspan="32">Key Seed Material (16 Bytes)
●
●
●</td></tr>
<tr><td colspan="32"></td></tr>
</table>

Table2.4B - Detailed Structure of an Onion Layer

The currently defined flags are as defined in the table below. A one signifies enabled while a zero signifies disabled.

Fields	Zero	Flags			Version			
Bit number	7	6	5	4	3	2	1	0
Bit Description	0	Tunnel	Replayable	In Band Signaling	Major Version	Minor Version		
Bit values	0	0/1	0/1	0/1	2-3	0-7		

Table 2.4C - Onion Flags and Version Information Layout

The forward and backward cryptographic function indices are encoded as shown in the following table.

Index	Algorithm	Key Length
0	NULL	N/A – no encryption
1	DES Stream	8 bytes (56 bits actual)

Index	Algorithm	Key Length
2	RC4	16 bytes
3	IDEA	8 bytes
4	Illegal	N/A
5-15	Reserved	N/A

Table2.4D - Cryptographic Function Index Definitions

Additionally, when a anonymous circuit is created additional data packaged in DATA cells immediately follows the onion. In order to unify the description of onions for the various types of connections the onion proper and this additional connection info data will be collectively referred to as the onion.

There are six types of additional onion data that can be sent in the payload of DATA cells. These are the following:

a) The first DATA cell payload onion data is known as the responder header and contains the following fields in the DATA cell payload:

Field Name	Length (bytes)	Description
Version	4 bits	The version of the onion network software
Flags	4 bits	Flags for circuit options
Protocol	1	The application protocol that this circuit will be using depending on the protocol the user is using (i.e. the application proxy protocol)
Retry Count	1	How many times the responder proxy or reply onion processor will attempt to connect to the connection acceptor

Table 2.4E - Responder Header Payload Data Fields

The structure of the DATA cell payload having a responder header is as follows.

<table>
<tr><td></td><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td>0</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Version</td><td colspan="4">Flags</td><td colspan="8">Protocol</td><td colspan="8">Retry Count</td><td colspan="8">Reserved</td></tr>
</table>

Table 2.4F - Structure of a DATA Cell Payload containing a Responder Header

There is currently only a single valid flag and that is the reply circuit indicator. When this flag is set it indicates that the circuit is a reply circuit.

For reply circuits only the version & flag fields of the responder header apply.

b) The second piece of data that can be sent is a destination host address including the port. This is the IP address and port of the destination host specified in the format delineated in the responder header. This address info appears after the responder header in non-reply circuits and after the reply onion processor header in reply circuits.

c) The third piece of information is the reply onion processor address and port. This information appears for reply circuits only. This is the address and port of the appropriate reply onion processor that this circuit should be assigned to, in the format specified in the responder header.

d) The fourth third piece of information is the key seed material for the keys used in a reply connection. This enables the reply onion proxy to obtain the keys to the anonymous circuit without having to store them for an indefinite period of time.

e) A fifth piece of information is the reply onion processor header. This only appears in reply circuits and contains the following fields:

Field Name	Length (bytes)	Description
Version	4 bits	Version of the onion routing software
Flags	4 bits	circuit option flags
Protocol	1	Application protocol
Retry Count	1	Number of retries for connection to connection acceptor
Address Format	1	Format that destination host address appears in

Table 2.4G - Reply Onion Processor Header Fields

It should be noticed that the reply onion processor header contains all of the information that appears in the responder header for non-reply circuits.

f) A sixth piece of information that can be sent after the first onion is one or more additional onions that will be used to define a tunneled route. These additional onions will be described further in the discussion of onion router core cell handling for tunneled routes.

2.4.1 Forward Onions

Forward onions are used to create anonymous circuits that will be used for ordinary connections. A forward onion consists of an onion proper, a responder header in the payload of a DATA cell, and destination host address and port in a the payload of a DATA cell.

2.4.2 Reply Onions

Reply onions are used to create anonymous circuits that will be used for reply connections. A reply onion consists of an onion proper, a responder header that contains an indicator that the onion is a reply onion, the reply onion processor address and port, the reply onion processor header, the destination host information, and finally the key seeds. The reply onion processor header, the destination host information, and the key seeds are all encrypted under the reply onion processors public key.

2.4.3 Replayable Reply Onions

Replayable reply onions are used to create anonymous circuits that will be used for replayable reply connections. A replayable reply onion contains an onion proper, with a bit set in the flags field indicating that it is a reply onion, a responder header with a bit set in the flags field indicating that the onion is a reply onion, the address and port of the reply onion processor, a reply onion processor header, the destination host address and port and finally the key seed material. The reply onion processor header, the destination host information, and the key seeds are all encrypted under the reply onion processors public key.

2.4.4 Rekeying Onions

Rekeying onions are used to distribute new symmetric keys for a replayable reply connection. A rekeying onion consists of 12 layers with layered encryption similar to an ordinary onion. Rekeying onions differ from ordinary onions in their structure and in that rekeying onions are encrypted using only symmetric cryptography. Public key cryptography is not used for encrypting layers of rekeying onions. A layer of a rekeying onion contains the following fields.

Field Name	Size (bytes)	Description
Flags	2	Currently the only defined flag is the tunnel indicator flag.
Key seed material	16	Seed for generating the new forward and reverse circuit encryption keys

Table 2.4.4A - Data Fields in a Rekeying Onion Layer

The structure of a rekeying onion layer is as follows.

<table>
<tr><th></th><th>3</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th>2</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th>1</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th></tr>
<tr><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="16">Flags</td><td colspan="16">Reserved</td></tr>
<tr><td colspan="32">Key Seed Material (16 bytes)
●
●
●</td></tr>
</table>

Table 2.4.4B _ Structure of a Rekeying Onion Layer

<table>
<tr><th></th><th></th><th></th><th></th><th></th><th>1</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th></tr>
<tr><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="15">Reserved</td><td>TI</td></tr>
<tr><td colspan="16"></td></tr>
<tr><td colspan="16"><b>Table 2.4.4C - Detail of Rekeying Onion Layer Flags Field</b></td></tr>
</table>

Note that TI stands for “Tunnel Indicator”.

Since a rekeying onion layer consists of 20 bytes a DATA cell, which is the cell type that rekeying onions are packaged in, contains 6 layers. Thus it takes two DATA cells to fit an entire rekeying onion.

The use of a rekeying onion is as follows. A replayable reply circuit has been set up and is desired to be rekeyed before passing application data over it. At this point all of the circuit participants have anonymous circuit data structures set up so that cells can be propagated in either direction across the circuit. Before sending the one byte acknowledgement that signifies circuit setup completion in an ordinary circuit the reply onion processor sends a rekeying onion down the reverse circuit back towards the initiator to distribute the new circuit crypto keys to the circuit participants. Each layer of the rekeying onion must be encrypted such that when it is decrypted by each onion router core that core sees the keyseed material for its new keys in the clear. The detailed algorithm for building a rekeying onion is covered in section ??? on the reply onion builder portion of the application proxy.

Each onion router core processes the rekeying onion as follows.

a) Receive all of the DATA cells comprising the first rekeying onion.
b) Decrypt the entire rekeying onion with the existing symmetric encryption key for the reverse circuit.
c) Obtain it’s new keyseed material from the topmost layer of the onion and note the tunnel indicator bit.
d) Hash the keyseed with SHA1 once to obtain the forward key and again to obtain the reverse key.
e) If the tunnel indicator bit is set, discard the DATA cells containing the first rekeying onion and skip to step 9.
f) If the tunnel indicator bit is not set, remove the topmost layer of the rekeying onion and append one rekeying onion layer’s worth of random bytes to the end of the onion.
g) Repackage the onion in two DATA cells.
h) Forward the remaining DATA cells containing the onion (including the first if the tunnel indicator bit was not set) to the next hop of the reverse circuit.
i) Rekey its encryption engines with the new keys and mark the circuit as up.

2.4.5 Tunneling Onions

Tunneling onions are used to create anonymous circuits that will be "tunneled". A tunneled circuit consists of more hops than will fit into the largest allowable onion size. A tunneled onion consists of two or more onions proper, a responder header, and destination host information. The last layer of all but the last onion contains a bit in the flags field indicating that tunneling will occur at this hop and that the next N DATA cells should be repackaged as CREATE cells and forwarded to the host whose address and port appear in the next hop field, where N is the number of cells required to package an onion (currently 2).

2.5 Global Identifiers

In order to avoid tying the identification of specific instances of network sub-systems to a particular network addressing scheme a subset of the network sub-systems will be assigned an individual global identifier (GID). This GID will be an unsigned short thus allowing the unique identification of 64K network sub-system instances. Currently the only network sub-system assigned a GID is the onion router cores.

The network administrator is responsible for the assignment of GIDs to onion router cores and is thus responsible for ensuring their uniqueness.

There is no central repository for GIDs that is accessible to network sub-systems at run time. The GIDs of a network sub-systems neighbors appear in that sub-systems configuration file. It is the responsibility of the sub-system administrator to obtain these GIDs from the network administrator prior to running the sub-system.

3. Onion Router Core Requirements

This section details the requirements for the onion router core sub-system.

3.1 Communications Requirements

3.1.1 Onion Router Core Peers

Onion router cores will maintain virtual pipe connections to neighboring sub-systems known as the cores peers or neighbors.

3.1.1.1 Onion Router Core Peer Sub-Systems

Onion router core peers will be the following:

a) one or more onion router cores
b) optionally one or more input funnel
c) one or more cryptographic processors
d) optionally one or more application proxies
e) one output funnel
f) one database engine

3.1.1.2 Master/Slave Relationships Between Core Peer Sub-Systems

As described in section 2.3 each sub-system at one end of a virtual pipe has a master/slave relationship with the sub-system on the other end of that pipe. The master/slave relationships between an onion router core and its peer sub-systems is as follows:

a) The master/slave relationship between peer onion router cores will be determined from the configuration file of each core. It is the responsibility of the core administrators to ensure that the two ends of a virtual pipe agree on their relationship.
b) The onion router core is master of all input funnel connections
c) The onion router core is master of the cryptographic processor connection
d) The onion router core is master of all application proxy connections
e) The onion router core is master of the output funnel connection
f) The onion router core is master of the database engine connection

3.1.1.3 Acceptable Cell Types for Given Peer Connections

The following table lists the acceptable cell types for each of the different virtual pipe connections to an onion router core sub-system.

Sending Sub-System	Receiving Sub-System	Valid Cell Types
Onion Router Core	Onion Router Core	CREATE, DATA, DESTROY, PADDING, DB_UPDATE, DB_QUERY, ACI_RANGE_ALLOC
Onion Router Core	Input Funnel	DATA, DESTROY, PADDING, DB_UPDATE, ACI_RANGE_ALLOC
Onion Router Core	Output Funnel	DATA, DESTROY, PADDING, ACI_RANGE_ALLOC
Onion Router Core	Database Engine	PADDING, DB_QUERY, DB_UPDATE
Onion Router Core	Cryptographic Processor	CREATE, DESTROY, PADDING
Onion Router Core	Application Proxy	DATA, DESTROY, PADDING, DB_UPDATE, ACI_RANGE_ALLOC
Input Funnel	Onion Router Core	CREATE, DATA, DESTROY, PADDING, DB_QUERY, ACI_RANGE_ALLOC
Output Funnel	Onion Router Core	DATA, DESTROY, PADDING, ACI_RANGE_ALLOC
Database Engine	Onion Router Core	PADDING, DB_QUERY, DB_UPDATE
Cryptographic Processor	Onion Router Core	CREATE, DESTROY, PADDING, ONION_INFO
Application Proxy	Onion Router Core	CREATE, DATA, DESTROY, PADDING, DB_QUERY, ACI_RANGE_ALLOC

Table 3.1.1.3 - Valid Cell Types for Onion Router Core Peer Connections

The receipt of any invalid cell type along a given virtual pipe is an error condition and will result in the destruction of the anonymous circuit upon which the errant cell was received.

3.1.2 Onion Router Listener Socket

Onion router cores will passively listen for active connections from its peers. This is to enable peers whose relation to a given core is master to actively connect to the core and also enables virtual pipe recovery in the event of a pipe error. The port that the core will listen for connections on will be well known to the core's peers and should probably be a port number that is globally used for pipe connection request across every network element.

3.1.3 Onion Router Core Virtual Pipe Connections

As discussed in section 3.1.1.2 onion router cores have a master/slave relationship with each of their peers. During the initialization of an onion router core a virtual pipe connection will be established between the onion router core and each of its peers. The master/slave relationship determines how this connection is established at startup.

On the virtual pipe connections where the onion router core is the master that core will actively connect to its slave peer by connecting to that slave peers well known listening socket and establishing a new connection. On the virtual pipe connections where the onion router core is a slave the core will passively listen for a connection request from that master peer.

In both cases the onion router core will fork off a child process to handle the pipe initialization so that the key exchange protocol does not cause too much of a disruption of active traffic through the core.

In both cases the master of the connection will initiate the key exchange protocol and will assign the valid ACI range for the virtual pipe.

3.2 Cell Processing Requirements

An onion router core receives cells from the virtual pipes it is connected to. There are essentially four actions that an onion router core can perform on a cell. It can ignore and drop that cell, it can forward that cell further along its anonymous circuit, it can process that cell to obtain information from it, and it can respond to a query contained in that cell.

As described in section 2.1.4 certain cell types are circuit oriented while others are pipe oriented. Processing of circuit oriented cells will depend upon the state of the circuit while processing of pipe oriented cells will depend upon the state of the pipe.

Further, an onion router core can autonomously create the PADDING and DB_UPDATE cell types and forward them to the appropriate destination. During virtual pipe initialization a master onion router core can also autonomously create the ACI_RANGE_ALLOC cell and forward it to the slave peer.

The following sections discuss in detail the cell processing requirements for given types of cells on given types of connections.

3.2.1 Ordinary Circuits

This section discussed processing of cells on ordinary connections. Note that the awaiting_rekey state is never reached in an ordinary connection and hence does not appear in this section.

3.2.1.1 Circuit Oriented Cells

This section describes the specific cell processing for various circuit oriented cell types dependent upon the state of the circuit.

3.2.1.1.1 CREATE cells

CREATE cells are used to create anonymous circuits.

3.2.1.1.1.1 Processing of a CREATE cell in the down state

Upon receipt of a CREATE cell for a circuit in the down state the onion router core will create a anonymous circuit structure for the forward and reverse circuits, mark the circuit state as created, and forward the cell to the Cryptographic Processor.

3.2.1.1.1.2 Processing of a CREATE cell in the created state

Upon receipt of a CREATE cell for a circuit in the created state from any network element except the Cryptographic Processor the onion router core will forward that cell to the Cryptographic Processor.

If the CREATE cell is received from the Cryptographic Processor then this is a cell containing a portion of the processed onion used to create this anonymous circuit that is ready to be forwarded to the next hop. This cell will be forwarded to the next hop host as specified in the ONION_INFO cell for this circuit.

If this CREATE cell is the last one that will be received from the Cryptographic Processor for this circuit then the circuit state will be marked as up immediately after this cell has been written into the mix queue of the virtual pipe of the next hop.

3.2.1.1.1.3 Processing of a CREATE cell in the up state

Receipt of a CREATE cell for a circuit in the up state is an error condition. The onion router core will forward DESTROY cells along the forward and reverse circuit and will mark the circuit as active_kill. This event will be logged as a protocol failure.

3.2.1.1.1.4 Processing of a CREATE cell in the pending state

Receipt of a CREATE cell for a circuit in the pending state is an error condition. However, the onion router core has already sent DESTROY cells along this circuit to destroy it so this cell is just dropped. This event will be logged as a protocol failure.

3.2.1.1.1.5 Processing of a CREATE cell in the active_kill state

Processing of CREATE cells received on a circuit in the active_kill state is identical to processing done when the circuit is in the pending state. This event will be logged as a protocol failure.

3.2.1.1.2 DATA cells

DATA cells are used to carry user data through the onion network.

3.2.1.1.2.1 Processing of a DATA cell in the down state

Receipt of a DATA cell for a circuit in the down state is an error condition. The onion router core will ignore and drop the cell. This event will be logged as a protocol failure.

3.2.1.1.2.2 Processing of a DATA cell in the created state

Upon receipt of a DATA cell for a circuit in the created state that cell will be locally buffered until the circuit enters the up state. At that time the cell will be forwarded to the next hop in the circuit, or to the output funnel if the receiving core is the last hop in the circuit.

3.2.1.1.2.3 Processing of a DATA cell in the up state

Data cells that are received on a circuit in the up state will be forwarded to the next hop in the circuit or to the output funnel if the receiving core is the last hop in the circuit.

If the anonymous circuit was created with the signaling option turned on the onion router core must first check that the payload of the DATA cell does not contain the signaling tag for that core. If it does, then the DATA cell is considered as a signaling cell, the signal information contained in the second half of the DATA cell payload will be interpreted and acted upon and the cell will be dropped instead of forwarded.

3.2.1.1.2.4 Processing of a DATA cell in the pending state

While the receipt of a DATA cell on a circuit in the pending state is not an error, it can happen if the DATA cell is sent before a DESTROY message is received at the sending core, these DATA cells will be ignored and dropped.

3.2.1.1.2.5 Processing of a DATA cell in the active_kill state

Processing of DATA cells on a circuit in the active_kill state is identical to the processing done when the circuit is in the pending state.

3.2.1.1.3 DESTROY cells

DESTROY cells are used to instruct network elements to take down a anonymous circuit and to acknowledge receipt of a DESTROY cell.

3.2.1.1.3.1 Processing of a DESTROY cell in the down state

Receipt of a DESTROY cell in the down state is an error condition. The cell will be ignored and dropped. This event will be logged as a protocol failure.

3.2.1.1.3.2 Processing of a DESTROY cell in the created state

Upon receipt of a DESTROY cell on a circuit in the created state the onion router core will

a) forward the DESTROY cell back down the reverse circuit to the network element that sent the DESTROY cell. This serves as acknowledgment of receipt of the DESTROY cell.
b) forward a DESTROY cell to the Cryptographic Processor and mark
c) the circuit state as pending.

3.2.1.1.3.3 Processing of a DESTROY cell in the up state

Upon receipt of a DESTROY cell on a circuit in the up state the onion router core will

a) forward the DESTROY cell back down the reverse circuit to the network element that sent the DESTROY cell. This serves as acknowledgment of receipt of the DESTROY cell.
b) forward a DESTROY cell to the next hop on the circuit or to the output funnel if the receiving core is the last hop on the circuit.
c) mark the circuit state as pending

3.2.1.1.3.4 Processing of a DESTROY cell in the pending state

Upon receipt of a DESTROY cell on a circuit in the pending state the onion router core will free the anonymous circuit data structures and implicitly mark the circuit state as down.

3.2.1.1.3.5 Processing of a DESTROY cell in the active_kill state

Upon receipt of a DESTROY cell on a circuit in the active_kill state the onion router core will mark the circuit state as pending.

3.2.1.1.4 Processing of ONION_INFO cells

ONION_INFO cells are used to send information contained in an onion from the cryptographic processor to the onion router core.

3.2.1.1.4.1 Processing of an ONION_INFO cell in the down state

The receipt of an ONION_INFO cell in the down state is an error condition. The cell will be ignored and dropped. This event will be logged as a protocol failure.

3.2.1.1.4.2 Processing of an ONION_INFO cell in the created state

Upon receipt of an ONION_INFO while in the created state the onion router core will complete the initialization of the anonymous circuit structures to prepare for the forwarding of the CREATE cells that will be forthcoming from the cryptographic processor.

There are three bits in the flags field of an ONION_INFO cell that affect the properties of the created circuit. The first bit is the tunnel route enable bit which indicates to the core that the circuit will be tunneled starting at this core. This will be discussed further in section 3.2.4 on tunneled circuits.

The second is the replay enable bit which specifies that the circuit being created will be a replayable reply circuit.

The third bit is the signaling enable bit which enables circuit master to core signaling as discussed in section 2.2.6. If this bit is present then the onion router core must check all valid DATA cells for the signaling tag as described in that section.

3.2.1.1.4.3 Processing of an ONION_INFO cell in the up state

The receipt of ONION_INFO cell on a anonymous circuit in the up state is an error condition The cell will be ignored and dropped. This event will be logged as a protocol failure.

3.2.1.1.4.4 Processing of an ONION_INFO cell in the pending state

The receipt of an ONION_INFO cell on a anonymous circuit in the pending state is an error condition. The cell will be ignored and dropped. This event will be logged as a protocol failure.

3.2.1.1.4.5 Processing of an ONION_INFO cell in the active_kill state

Receipt of an ONION_INFO cell should never occur because a onion router core cannot enter the active_kill state from the created state because it has not constructed enough of the circuit to be able to forward DESTROY cells down both sides of it. Nevertheless if the cryptographic processor forwards an ONION_INFO cell to the core while it's in the active_kill state the cell will be ignored and dropped. This event will be logged as a protocol failure.

This might be cause for the onion router core to reset the cryptographic processor if we decided that it has that capability.

3.2.1.2 Pipe Oriented Cells

This section describes the processing of the pipe oriented cells.

3.2.1.2.1 PADDING cells

PADDING cells are injected by onion router cores to confuse the timing and volume signatures of virtual pipe. They carry no information.

3.2.1.2.1.1 Processing of a PADDING cell

Upon receipt of a PADDING cell the onion router core will ignore and drop the cell.

3.2.1.2.2 DB_UPDATE cells

DB_UPDATE cells are used to propagate onion network topology and public key information.

3.2.1.2.2.1 Processing of a DB_UPDATE cell

Upon the receipt of a DB_UPDATE cell from any network element other than the database engine the onion router core will forward that cell to the database engine. Additionally the onion router core will alter the ACI field of the DB_UPDATE cell being forwarded by setting it to the ACI of the reverse circuit of the existing ACI.

Upon the receipt of a DB_UPDATE cell from the database engine the onion router core will forward that cell along the circuit indicated in the cell's ACI field. This will be the reverse circuit of the circuit that the cell was received on and thus the any answer from the topology unit will be forwarded back to the network element sending the query.

3.2.1.2.3 DB_QUERY cells

DB_QUERY cells are used to query a database engine regarding current onion network topology and public key information. DB_QUERY cells are also used by a database engine to query a onion router core for it's current link state or its public key.

DB_QUERY cells can only be received on virtual pipes either to a database engine or to a sub-system that can be a primary peer to a database engine, i.e. an onion router core, an input funnel, an output funnel, or an application proxy. Any DB_* cells received from sub-systems other than the forementioned constitute and error condition. The errant cells will be ignored and dropped and the event logged as a protocol failure.

3.2.1.2.3.1 Processing of a DB_QUERY cell

Processing of a DB_QUERY cell depends upon the type of query as specified by the DB_QUERY_TYPE field in the cell payload.

3.2.1.2.3.1.1 Local Topology Query

A DB_QUERY cell for local topology is processed by creating a one or more DB_UPDATE cells with the DB_UPDATE_TYPE field set to DB_UPDATE_TYPE_TOPO and the DB_UPDATE_DATA fields containing link update information for each of the virtual pipes maintained by the onion routing core. For each link update the DB_UPDATE_TYPE_TOPO_ID1 field will be set to the global identifier of the onion router core, the DB_UPDATE_TYPE_TOPO_ID2 field will be set to the global identifier of the far end peer if that peer is an onion router core, otherwise it will be set to zero. The DB_UPDATE cell is sent back to the database engine on virtual pipe #0, on the anonymous circuit with the same number as the virtual pipe number (as seen by the core sending the update) of the pipe whose link status is being updated.

This cell is only valid when received from the database engine sub-system. If it is received on any virtual pipe other than that to the database engine the cell will be ignored and dropped and the event will be logged as a protocol failure.

3.2.1.2.3.1.2 Global Topology Query

A DB_QUERY cell for global topology, received from the database engine sub-system is processed by forwarding the cell along the proper circuit. The proper circuit is determined by first noting the ACI the cell was received on, call this N, and then forwarding the cell on the circuit reserved for DB_ * cells on virtual pipe N. This database specific circuit is always the lowest numbered circuit in the ACI range of the virtual pipe. Thus if the global topology query was received on circuit 2, and virtual pipe 2 has a lowest numbered ACI of 256, then the cell is forwarded onto virtual pipe #2, circuit #256.

If the DB_QUERY cell is received on a circuit with a number higher than the highest numbered virtual pipe then the cell is ignored and dropped and the event is logged as a protocol failure. If the DB_QUERY cell is received on a circuit for which the corresponding virtual pipe is down, then the cell is ignored and dropped, the event is logged as a protocol failure, and a specific topology update will be sent to the database engine indicating that the virtual pipe is down. This event is logged as a database error event.

If the DB_QUERY cell for global topology is received on a virtual pipe other than that to the database engine then the cell is forwarded to the database engine on anonymous circuit #0 of virtual pipe #

3.2.1.2.3.1.3 Specific Topology Query

The DB_QUERY cell for specific topology is processed identically to the DB_QUERY cell for global topology.

3.2.1.2.3.1.4 Local Public Key Query

A DB_QUERY cell for a local public key is processed by formatting a series of DB_UPDATE cells with the onion router core's public key certificate and forwarding those cells back to the database engine. The format of the DB_UPDATE cell, DB_UPDATE_TYPE_KEY, update type is described in section ??? on DB_* cells.

A DB_QUERY cell is only valid when received from the database engine. If the cell is received from any other sub-system it is ignored and dropped and logged as a protocol error.

3.2.1.2.3.1.5 Global Public Key Query

A DB_QUERY cell for a global public key update is processed depending upon what sub-system it was received from. If the query cell was received from the database engine then it is forwarded on the circuit as appropriate for the circuit it was received on. As noted above the cell will be forwarded on the database circuit on the virtual pipe with the corresponding number as the circuit that the cell was received on.

If the DB_QUERY cell was received from an application proxy, input funnel, onion router core, or output funnel then that cell will be forwarded to the database engine on virtual pipe #0, on the anonymous circuit number equal to the virtual pipe number that the cell was originally received on.

3.2.1.2.3.1.6 Specific Public Key Query

DB_QUERY cells for specific public key updates are handled identically to DB_QUERY cells for global public key updates.

3.2.1.2.3.2 ACI_RANGE_ALLOC cells

ACI_RANGE_ALLOC cells are used to negotiate the ACI range for a given virtual pipe. The protocol for ACI range negotiation is detailed in section ????

3.2.1.2.3.3 Processing of a ACI_RANGE_ALLOC cell

The processing of a ACI_RANGE_ALLOC cell depends upon whether the onion router core is the master or the slave of the virtual pipe that the cell is received on, and on the sub-command of the ACI_RANGE_ALLOC cell.

3.2.1.2.3.3.1 VP Master Processing of a ACI_RANGE_ALLOC cell

This section details the ACI_RANGE_ALLOC cell processing for a onion router core that is master of a given virtual pipe.

3.2.1.2.3.3.1.1 Processing of a ACI_RANGE_ALLOC Assignment cell

The master of a virtual pipe should never receive a ACI_RANGE_ALLOC assignment cell. The cell will be ignored and dropped and the event logged as a protocol failure.

3.2.1.2.3.3.1.2 Processing of a ACI_RANGE_ALLOC ACK cell

Receipt of a ACI_RANGE_ALLOC ACK cell by a virtual pipe master indicates that the pipe slave has accepted the assigned ACI range. The cell is processed by terminating the ACI range allocation protocol and reporting to a higher level routine that the protocol terminated successfully, or alternatively by simply continuing with the process of virtual pipe initialization.

3.2.1.2.3.3.1.3 Processing of a ACI_RANGE_ALLOC NAK cell

Receipt of a ACI_RANGE_ALLOC NAK cell by a virtual pipe master indicates that the pipe slave was not able to accept the last proposed ACI range assignment. This cell is processed by selecting a different range, formatting a ACI_RANGE_ALLOC assignment cell specifying this range, and sending that cell to the virtual pipe far end peer. Alternatively if this is the (N+1)th ACI_RANGE_ALLOC Assignment cell that the onion router core has received and the core is configured to only try N iterations of the ACI range allocation protocol then the core will terminate the ACI range allocation protocol by formatting a ACI_RANGE_ALLOC Abort cell and sending it to the far end peer. It will then either report the protocol failure to a higher level routine or it will cease the process of virtual pipe initialization.

3.2.1.2.3.3.1.4 Processing of a ACI_RANGE_ALLOC Abort cell

The master of a virtual pipe should never received a ACI_RANGE_ALLOC Abort cell. The cell will be ignored and dropped and the event logged as a protocol failure.

3.2.1.2.3.3.2 VP Slave Processing of a ACI_RANGE_ALLOC cell

This section details the ACI_RANGE_ALLOC cell processing for an onion router core that is slave of a given virtual pipe.

3.2.1.2.3.3.2.1 Processing of a ACI_RANGE_ALLOC Assignment cell

When an onion router core that is slave of a given virtual pipe receives a ACI_RANGE_ALLOC Assignment cell it first checks that the assignment does not conflict with any other range assignment for other virtual pipes it maintains. If there are no range conflicts caused by this assignment the cell’s sub-command is changed from ASSIGN to ACK and the cell is forwarded back to the pipe master. The assigned ACI range is then set into the virtual pipe data structure.

If the range assignment does conflict with an existing range then the cell sub-command is changed to NAK and the cell is forwarded back to the master.

3.2.1.2.3.3.2.2 Processing of a ACI_RANGE_ALLOC ACK cell

The slave of a virtual pipe should never receive a ACI_RANGE_ALLOC ACK cell. The cell will be ignored and dropped and the event logged as a protocol failure.

3.2.1.2.4 Processing of a ACI_RANGE_ALLOC NAK cell

The slave of a virtual pipe should never receive a ACI_RANGE_ALLOC NAK cell. The cell will be ignored and dropped and the event logged as a protocol failure.

3.2.1.2.4.1.1.1 Processing of a ACI_RANGE_ALLOC Abort cell

Upon receipt of a ACI_RANGE_ALLOC Abort cell the slave of a virtual pipe will forward the cell unmodified back to the master of the pipe. The slave will then terminate the ACI range allocation protocol and perform any data structure clean up as required.

3.2.2 Reply Circuits

The onion router core cell processing for reply circuits is identical to that in ordinary circuits.

3.2.3 Replayable Reply Circuits

Replayable reply circuits must be rekeyed before their use for application data. This introduces the awaiting_rekey state which affects the way in which DATA cells are processed.

3.2.3.1 Circuit Oriented Cells

This section describes the specific cell processing for various circuit oriented cell types dependent upon the state of the circuit.

3.2.3.1.1 CREATE cells

CREATE cells are used to create anonymous circuits.

3.2.3.1.1.1 Processing of a CREATE cell in the down state

Upon receipt of a CREATE cell for a circuit in the down state the onion router core will create a anonymous circuit structure for the forward and reverse circuits, mark the circuit state as created, and forward the cell to the Cryptographic Processor.

3.2.3.1.1.2 Processing of a CREATE cell in the created state

Upon receipt of a CREATE cell for a circuit in the created state from any network element except the Cryptographic Processor the onion router core will forward that cell to the Cryptographic Processor.

If the CREATE cell is received from the Cryptographic Processor then this is a cell containing a portion of the processed onion used to create this anonymous circuit that is ready to be forwarded to the next hop. This cell will be forwarded to the next hop host as specified in the ONION_INFO cell for this circuit.

If this CREATE cell is the last one that will be received from the Cryptographic Processor for this circuit then the circuit state will be marked as awaiting_rekey immediately after this cell has been written into the mix queue of the virtual pipe of the next hop.

3.2.3.1.1.3 Processing of a CREATE cell in the awaiting_rekey state

Receipt of a CREATE cell in the awaiting_rekey state is an error condition. The onion router core will forward DESTROY cells along the forward and reverse circuit and will mark the circuit as active_kill. The event will be logged as a protocol failure.

3.2.3.1.1.4 Processing of a CREATE cell in the up state

Receipt of a CREATE cell for a circuit in the up state is an error condition. The onion router core will forward DESTROY cells along the forward and reverse circuit and will mark the circuit as active_kill. The event will be logged as a protocol failure.

3.2.3.1.1.5 Processing of a CREATE cell in the pending state

Receipt of a CREATE cell for a circuit in the pending state is an error condition. However, the onion router core has already sent DESTROY cells along this circuit to destroy it so this cell is just dropped. The event will be logged as a protocol failure.

3.2.3.1.1.6 Processing of a CREATE cell in the active_kill state

Processing of CREATE cells received on a circuit in the active_kill state is identical to processing done when the circuit is in the pending state. The event will be logged as a protocol failure.

3.2.3.1.2 DATA cells

DATA cells are used to carry user data through the onion network.

3.2.3.1.2.1 Processing of a DATA cell in the down state

Receipt of a DATA cell for a circuit in the down state is an error condition. The onion router core will ignore and drop the cell. The event will be logged as a protocol failure.

3.2.3.1.2.2 Processing of a DATA cell in the created state

Upon receipt of a DATA cell for a circuit in the created state that cell will be locally buffered until the circuit enters the up state. At that time the cell will be forwarded to the next hop in the circuit, or to the output funnel if the receiving core is the last hop in the circuit.

3.2.3.1.2.3 Processing of a DATA cell in the awaiting_rekey state

DATA cells received in the awaiting_rekey state contain a rekeying onion used to rekey the circuit. DATA cells received while in the awaiting_rekey state are first buffered until an entire rekeying onion has been received. As a rekeying onion contains 12 layers this will be after two DATA cells have been received. The rekeying onion is processed as follows:

a) The entire onion is extracted from the payloads of the DATA cells and decrypted using the existing symmetric key stream for the reverse circuit.
b) The topmost layer is extracted from the plaintext of the onion. The keyseed material stored and the tunnel indicator bit value noted.
c) If the tunnel indicator bit was set then the DATA cells containing the onion are discarded after obtaining the keyseed material and steps 4 and 5 are ski
d) If the tunnel indicator bit was not set then the topmost layer of the onion is removed and a layer of random bytes is appended to the remaining onion.
e) The processed onion is repacked into DATA cells and forwarded to the next hop in the reverse circuit.
f) The keyseed material is hashed using SHA1. Once to obtain the forward key and again to obtain the reverse key. The cryptographic engines are then rekeyed using the new keys.
g) The circuit is marked as up. (a awaiting_rekey to up state transition)
h) Any further DATA cells (which may contain additional rekeying onions) are then processed as usual for a replayable reply circuit in the up state.

3.2.3.1.2.4 Processing of a DATA cell in the up state

Data cells that are received on a circuit in the up state will be forwarded to the next hop in the circuit or to the output funnel if the receiving core is the last hop in the circuit, or the input funnel or application proxy if the receiving core is the last hop of the reverse circuit.

3.2.3.1.2.5 Processing of a DATA cell in the pending state

While the receipt of a DATA cell on a circuit in the pending state is not an error, it can happen if the DATA cell is sent before a DESTROY message is received at the sending core, these DATA cells will be ignored and dropped.

3.2.3.1.2.6 Processing of a DATA cell in the active_kill state

Processing of DATA cells on a circuit in the active_kill state is identical to the processing done when the circuit is in the pending state.

3.2.3.1.3 DESTROY cells

DESTROY cells are used to instruct network elements to take down a anonymous circuit and to acknowledge receipt of a DESTROY cell.

3.2.3.1.3.1 Processing of a DESTROY cell in the down state

Receipt of a DESTROY cell in the down state is an error condition. The cell will be ignored and dropped. This event will be logged as a protocol failure.

3.2.3.1.3.2 Processing of a DESTROY cell in the created state

Upon receipt of a DESTROY cell on a circuit in the created state the onion router core will

a) forward the DESTROY cell back down the reverse circuit to the network element that sent the DESTROY cell. This serves as acknowledgment of receipt of the DESTROY cell.
b) forward a DESTROY cell to the Cryptographic Processor and mark
c) the circuit state as pending.

3.2.3.1.3.3 Processing of a DESTROY cell in the awaiting_rekey state

Upon receipt of a DESTROY cell on a circuit in the awaiting_rekey state the onion router core will

a) forward the DESTROY cell back down the reverse circuit to the network element that sent the DESTROY cell. This serves as acknowledgment of receipt of the DESTROY cell.
b) forward a DESTROY cell to the next hop along the circuit or to the output funnel if this is the last hop in the circuit.
c) the circuit state as pending.

3.2.3.1.3.4 Processing of a DESTROY cell in the up state

Upon receipt of a DESTROY cell on a circuit in the up state the onion router core will

a) forward the DESTROY cell back down the reverse circuit to the network element that sent the DESTROY cell. This serves as acknowledgment of receipt of the DESTROY cell.
b) forward a DESTROY cell to the next hop on the circuit or to the output funnel if the receiving core is the last hop on the circuit.
c) mark the circuit state as pending

3.2.3.1.3.5 Processing of a DESTROY cell in the pending state

Upon receipt of a DESTROY cell on a circuit in the pending state the onion router core will free the anonymous circuit data structures and implicitly mark the circuit state as down.

3.2.3.1.3.6 Processing of a DESTROY cell in the active_kill state

Upon receipt of a DESTROY cell on a circuit in the active_kill state the onion router core will mark the circuit state as pending.

3.2.3.1.4 ONION_INFO cells

ONION_INFO cells are used to send information contained in an onion from the cryptographic processor to the onion router core.

3.2.3.1.4.1 Processing of an ONION_INFO cell in the down state

The receipt of an ONION_INFO cell in the down state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.3.1.4.2 Processing of an ONION_INFO cell in the created state

Upon receipt of an ONION_INFO while in the created state the onion router core will complete the initialization of the anonymous circuit structures to prepare for the forwarding of the CREATE cells that will be forthcoming from the cryptographic processor.

3.2.3.1.4.3 Processing of an ONION_INFO cell in the awaiting_rekey state

The receipt of ONION_INFO cell on a anonymous circuit in the awaiting_rekey state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.3.1.4.4 Processing of an ONION_INFO cell in the up state

The receipt of ONION_INFO cell on a anonymous circuit in the up state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.3.1.4.5 Processing of an ONION_INFO cell in the pending state

The receipt of an ONION_INFO cell on a anonymous circuit in the pending state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.3.1.4.6 Processing of an ONION_INFO cell in the active_kill state

Receipt of an ONION_INFO cell should never occur because a onion router core cannot enter the active_kill state from the created state because it has not constructed enough of the circuit to be able to forward DESTROY cells down both sides of it. Nevertheless if the cryptographic processor forwards an ONION_INFO cell to the core while it's in the active_kill state the cell will be ignored and dropped.

This might be cause for the onion router core to reset the cryptographic processor if we decided that it has that capability.

3.2.3.2 Pipe Oriented Cells

Pipe oriented cell processing for replayable reply circuits is identical to that of ordinary circuits.

3.2.4 Tunneled Circuits

Tunneled circuits are those that require more than a single onion to create the circuit. A tunneled circuit is only apparent to the circuit master and the last onion router core on each sub-circuit.
The fact that a route is tunneled is revealed to an onion router core via a bit in the flag field of the ONION_INFO cell received from its cryptographic processor. The fact that a route is tunneled affects how a onion router core processes DATA cells.

3.2.4.1 Circuit Oriented Cells

This section describes the specific cell processing for various circuit oriented cell types dependent upon the state of the circuit.

3.2.4.1.1 CREATE cells

CREATE cells are used to create anonymous circuits.

3.2.4.1.1.1 Processing of a CREATE cell in the down state

Upon receipt of a CREATE cell for a circuit in the down state the onion router core will create a anonymous circuit structure for the forward and reverse circuits, mark the circuit state as created, and forward the cell to the Cryptographic Processor.

3.2.4.1.1.2 Processing of a CREATE cell in the created state

Upon receipt of a CREATE cell for a circuit in the created state from any network element except the Cryptographic Processor the onion router core will forward that cell to the Cryptographic Processor.

If the CREATE cell is received from the Cryptographic Processor then this is a cell containing a portion of the processed onion used to create this anonymous circuit that is ready to be forwarded to the next hop. This cell will be forwarded to the next hop host as specified in the ONION_INFO cell for this circuit.

If this CREATE cell is the last one that will be received from the Cryptographic Processor for this circuit then the circuit state will be marked as awaiting_rekey immediately after this cell has been written into the mix queue of the virtual pipe of the next hop.

3.2.4.1.1.3 Processing of a CREATE cell in the awaiting_rekey state

Receipt of a CREATE cell in the awaiting_rekey state is an error condition. The onion router core will forward DESTROY cells along the forward and reverse circuit and will mark the circuit as active_kill. The event will be logged as a protocol failure.

3.2.4.1.1.4 Processing of a CREATE cell in the up state

Receipt of a CREATE cell for a circuit in the up state is an error condition. The onion router core will forward DESTROY cells along the forward and reverse circuit and will mark the circuit as active_kill. The event will be logged as a protocol failure.

3.2.4.1.1.5 Processing of a CREATE cell in the pending state

Receipt of a CREATE cell for a circuit in the pending state is an error condition. However, the onion router core has already sent DESTROY cells along this circuit to destroy it so this cell is just dropped. The event will be logged as a protocol failure.

3.2.4.1.1.6 Processing of a CREATE cell in the active_kill state

Processing of CREATE cells received on a circuit in the active_kill state is identical to processing done when the circuit is in the pending state.

3.2.4.1.2 DATA cells

DATA cells are used to carry user data through the onion network.

3.2.4.1.2.1 Processing of a DATA cell in the down state

Receipt of a DATA cell for a circuit in the down state is an error condition. The onion router core will ignore and drop the cell. The event will be logged as a protocol failure.

3.2.4.1.2.2 Processing of a DATA cell in the created state

Upon receipt of a DATA cell for a circuit in the created state that cell will be locally buffered until the circuit enters the awaiting_rekey state. At that time the cell will be forwarded to the next hop in the circuit, or to the output funnel if the receiving core is the last hop in the circuit.

Upon the forwarding of the last CREATE cell from the cryptographic processor to the next hop along the circuit the onion router core will mark the circuit in the awaiting_rekey state. At this time it will forward all DATA cells buffered while the circuit was in the created state.

Additionally if the received ONION_INFO cell denoted that this core is the last core on a sub-circuit then the first N buffered DATA cells (where N is the number of cells in an onion) will have their cell type changed to CREATE prior to forwarding along the circuit. This is so onion router cores along the next sub-circuit see CREATE cells containing an onion for setting up the circuit as is the usual case.

3.2.4.1.2.3 Processing of a DATA cell in the awaiting_rekey state

The DATA cell received in the awaiting_rekey state will hold the key seed material for the new symmetric keys for the anonymous circuit encryption. The onion router core will encrypt the cell, obtain it's key seed material from the first 16 bytes of the cell payload, move the remaining contents of the payload to the top of the cell payload, and pad the end of the payload with 16 bytes of random bits.

The onion router core will then apply SHA to the key seed material to obtain the new keys for the circuit. Since we do not need the additional symmetric key used for decrypting ordinary onions the key seed material need only be hashed twice rather than three times.

The onion router core will then rekey it's cryptographic engines with the new keys.

After padding the cell the onion router core will forward the cell to the next hop along the circuit.

After forwarding the cell the onion router core will mark the circuit as up.

3.2.4.1.2.4 Processing of a DATA cell in the up state

Data cells that are received on a circuit in the up state will be forwarded to the next hop in the circuit or to the output funnel if the receiving core is the last hop in the circuit.

3.2.4.1.2.5 Processing of a DATA cell in the pending state

While the receipt of a DATA cell on a circuit in the pending state is not an error, it can happen if the DATA cell is sent before a DESTROY message is received at the sending core, these DATA cells will be ignored and dropped. The event will be logged as a protocol failure.

3.2.4.1.2.6 Processing of a DATA cell in the active_kill state

Processing of DATA cells on a circuit in the active_kill state is identical to the processing done when the circuit is in the pending state.

3.2.4.1.3 DESTROY cells

DESTROY cells are used to instruct network elements to take down a anonymous circuit and to acknowledge receipt of a DESTROY cell.

3.2.4.1.3.1 Processing of a DESTROY cell in the down state

Receipt of a DESTROY cell in the down state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.4.1.3.2 Processing of a DESTROY cell in the created state

Upon receipt of a DESTROY cell on a circuit in the created state the onion router core will

a) forward the DESTROY cell back down the reverse circuit to the
b) network element that sent the DESTROY cell. This serves as
c) acknowledgment of receipt of the DESTROY cell.
d) forward a DESTROY cell to the Cryptographic Processor and mark
e) the circuit state as pending.

3.2.4.1.3.3 Processing of a DESTROY cell in the awaiting_rekey state

Upon receipt of a DESTROY cell on a circuit in the awaiting_rekey state the onion router core will

a) forward the DESTROY cell back down the reverse circuit to the network element that sent the DESTROY cell. This serves as acknowledgment of receipt of the DESTROY cell.
b) forward a DESTROY cell to the next hop along the circuit or to the output funnel if this is the last hop in the circuit.
c) mark the circuit state as pending.

3.2.4.1.3.4 Processing of a DESTROY cell in the up state

Upon receipt of a DESTROY cell on a circuit in the up state the onion
router core will

a) forward the DESTROY cell back down the reverse circuit to the network element that sent the DESTROY cell. This serves as acknowledgment of receipt of the DESTROY cell.
b) forward a DESTROY cell to the next hop on the circuit or to the output funnel if the receiving core is the last hop on the circuit.
c) mark the circuit state as pending

3.2.4.1.3.5 Processing of a DESTROY cell in the pending state

Upon receipt of a DESTROY cell on a circuit in the pending state the onion router core will free the anonymous circuit data structures and implicitly mark the circuit state as down.

3.2.4.1.3.6 Processing of a DESTROY cell in the active_kill state

Upon receipt of a DESTROY cell on a circuit in the active_kill state the onion router core will mark the circuit state as pending.

3.2.4.1.4 ONION_INFO cells

ONION_INFO cells are used to send information contained in an onion from the cryptographic processor to the onion router core.

3.2.4.1.4.1 Processing of an ONION_INFO cell in the down state

The receipt of an ONION_INFO cell in the down state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.4.1.4.2 Processing of an ONION_INFO cell in the created state

Upon receipt of an ONION_INFO while in the created state the onion router core will complete the initialization of the anonymous circuit structures to prepare for the forwarding of the CREATE cells that will be forthcoming from the cryptographic processor.

In tunneled circuits the onion router core that is the last hop of a sub-circuit will be informed of this by the fact that the tunneled route enable bit is set in the flags field of the ONION_INFO cell. When this bit is set the onion router core knows that the next N DATA cells (where N is the number of cells in an onion) are in fact the onion describing the next sub-circuit. Treatment of these data cells will be described in section 3.2.4.1.2.2.

3.2.4.1.4.3 Processing of an ONION_INFO cell in the awaiting_rekey state

The receipt of ONION_INFO cell on a anonymous circuit in the awaiting_rekey state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.4.1.4.4 Processing of an ONION_INFO cell in the up state

The receipt of ONION_INFO cell on a anonymous circuit in the up state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.4.1.4.5 Processing of an ONION_INFO cell in the pending state

The receipt of an ONION_INFO cell on a anonymous circuit in the pending state is an error condition. The cell will be ignored and dropped. The event will be logged as a protocol failure.

3.2.4.1.4.6 Processing of an ONION_INFO cell in the active_kill state

Receipt of an ONION_INFO cell should never occur because a onion router core cannot enter the active_kill state from the created state because it has not constructed enough of the circuit to be able to forward DESTROY cells down both sides of it. Nevertheless if the cryptographic processor forwards an ONION_INFO cell to the core while it's in the active_kill state the cell will be ignored and dropped. The event will be logged as a protocol failure.

This might be cause for the onion router core to reset the cryptographic processor if we decided that it has that capability.

3.2.5 Tunneled Reply Circuits

Cell processing on tunneled reply circuits is identical to that of tunneled circuits.

3.2.6 Tunneled Replayable Reply Circuits

Cell processing in tunneled replayable reply circuits is a combination of the cell processing that occurs in tunneled circuits and replayable reply circuits. Cell processing follows that of tunneled circuits up to the point of the created to up state transition, but rather than go directly to the up state the awaiting_rekey state is entered first. At that point cell processing is handled identically to replayable reply circuit cell processing.

3.2.7 Mixing

Prior to outputting the current batch of anonymous circuit cells to their associated virtual pipes, the cells from the anonymous circuits on a given virtual pipe are "mixed". Mixing consists of randomly selecting among the anonymous circuits on a virtual pipe to decide what cell is next to be written to that pipe. Additionally the decision may be made to send a padding cell that is not from any anonymous circuit.

The requirements for mixing are as follows:

a) The ordering of cells within a given anonymous circuit must be maintained when outputting cells to a virtual pipe;
b) The selection of from which anonymous circuit the next output cell comes from will be a random selection in that the selection will be based on a pseudo-random number obtained from a local random number source.
c) Padding cells will be used as necessary to attempt to closely match a chosen set of traffic statistics. These statistics will be locally averaged cell flow rates on a per virtual pipe basis.
d) In a given mix session all cells ready to be mixed, i.e. on a virtual pipe mix queue, will be mixed and output to a virtual pipe transmit buffer during that session.

3.3 Onion Router Core Error Handling Requirements

There are three types of errors that can occur on an onion router core. Local system errors, anonymous circuit errors, and virtual pipe errors.

3.3.1 Local System Error Handling

3.3.2 Anonymous circuit Error Handling

3.3.3 Virtual Pipe Error Handling

4. Application Proxy Requirements

4.1 Application Proxy Components

The application proxy consists of five components. These components are as follows.

Application Proxy Component	Component Function
Application Interface	Interface to application. Performs application specific connection initialization. Interprets onion routing error messages and converts these to application specific meaningful messages. Provides interface to reply onion builder.
Onion Interface	Interface to onion routing network. Encapsulates application data into cells. Handles sending of onions, additional onion data, etc.
Forward Onion Builder	Builds forward onions on behalf of the onion interface.
Reply Onion Builder	Builds reply onions upon request
Database Engine	Maintains topology and key information. Facilitates the building of onions.

Table 4.1 - Application Proxy Components

4.1.1 Application Interface

4.1.2 Onion Interface

4.1.3 Forward Onion Builder

4.1.4 Reply Onion Builder

4.1.5 Database Engine

4.2 Application Proxy Communications Requirements

4.2.1 Application Proxy Peers

Application proxies will maintain virtual pipe connections to neighboring sub-systems known as peers or neighbors.

4.2.1.1 Application Proxy Peer Sub-Systems

The application proxy peers are the following sub-systems.

a) optionally one input funnel
b) optionally one onion router core
c) one database engine

The application proxy must have either (a) or (b) as a peer. It cannot have both.

4.2.1.2 Master/Slave Relationships Between Core Peer Sub-Systems

a) the application proxy is slave to the input funnel.
b) the application proxy is slave to an onion router core
c) the application proxy is the master to the database engine.

4.2.1.3 Acceptable Cell Types for Given Peer Connections

The following table lists the acceptable cell types for each of the different virtual pipe connections to an onion router core sub-system.

Sending Sub-System	Receiving Sub-System	Valid Cell Types
Application Proxy	Input Funnel	CREATE, DATA, DESTROY, PADDING, DB_UPDATE, DB_QUERY, ACI_RANGE_ALLOC
Input Funnel	Application Proxy	DATA, DESTROY, PADDING, DB_UPDATE, DB_QUERY, ACI_RANGE_ALLOC

Table 4.2.1.3 - Valid Cell Types for Application Proxy Peer Connections

The receipt of any invalid cell type along a given virtual pipe is an error condition and will result in the destruction of the anonymous circuit upon which the errant cell was received.

4.2.2 Application Proxy Listener Sockets

The application proxy maintains a number of open sockets upon which it listens for connection requests. These listener sockets are used for two separate purposes, accepting connection requests from user applications, and accepting connection requests from the peer input funnel sub-system.

4.2.2.1 Application Connection Request Sockets

For each network application that is proxied the application proxy will listen for connection requests on a port that is well known to each application. These ports are those that the application user will set their application to use as proxy ports. Upon a application connection request the application proxy will accept that connection and assign a new socket for that request.

The lifetime of the application request sockets are the lifetime of the application proxy.

4.2.2.2 Input Funnel Connection Request Socket

The application proxy will listen on a well known port for connection requests from the input funnel sub-system. This is to facilitate virtual pipe recovery in the event of the input funnel sub-system going down. When the erring input funnel comes back up it will attempt to actively connect to the application proxy on this listener socket.

4.3 Application Proxy Cell Processing Requirements

4.3.1 Ordinary Connections

4.3.2 Reply Connections

4.3.3 Replayable Reply Connections

4.3.4 Tunneled Connections

4.3.5 Tunneled Reply Connections

4.3.6 Tunneled Replayable Reply Connections

4.4 Application Interface Requirements

4.4.1 HTTP Application

4.4.2 SMTP Application

4.4.3 Rlogin Application

4.4.4 Raw Socket Application

4.4.5 Reply Onion Builder Application

4.5 Application Proxy Error Handling Requirements

5. Input Funnel Requirements

An input funnel multiplexes multiple virtual pipe connections from application proxies onto a single virtual pipe connection to an onion router core. It likewise de-multiplexes the virtual pipe from the core to the multiple virtual pipes to the application proxies.

The requirements of the Input Funnel can be grouped into 3 categories, communications, cell processing, and error handling.

5.1 Input Funnel Communications Requirements

An input funnel communicates with its peers via virtual pipes. Virtual pipes are stream socket connections constrained to send data in the form of cells traveling on anonymous circuits. For more details on cells, anonymous circuits, and virtual pipes see section 2.0 of this document.

5.1.1 Input Funnel Peers

Input funnels will maintain virtual pipe connection to neighboring sub-systems known as the input funnels peers or neighbors. Input funnels can be "nested" so that a sequence of input funnels acts as essentially one conduit from an application proxy to an onion router core. Cells passing through the sequence of input funnels do so unmodified.

5.1.1.1 Input Funnel Peer Sub-systems

Input funnel peers will be the following:

a) an input funnel can maintain an optional virtual pipe connection to a peer onion router core.
b) an input funnel can maintain optional virtual pipe connections to one or more application proxies.
c) an input funnel will maintain a virtual pipe connection to an associated database engine.
d) an input funnel can maintain a optional virtual pipe connections to other input funnels.

5.1.1.2 Master/Slave Relationships Between Input Funnel Peer Sub-systems

As described in section 2.3 each sub-system at one end of a virtual pipe has a master/slave relationship with the sub-system at the other end. The master/slave relationships between the input funnel and its peers are as follows:

a) the input funnel will be the slave to the onion router core
b) the input funnel will be the master to each application proxy
c) the input funnel will be the master to the database engine
d) the master/slave relationship between peer input funnels will be specified in the configuration files of each.

5.1.1.3 Acceptable Cell Types for Given Peer Connections

The following table lists the acceptable cell types for each of the different virtual pipe connections to the input funnel sub-system.

Sending Sub-System	Receiving Sub-System	Valid Cell Types
Input Funnel	Onion Router Core	CREATE, DATA, DESTROY, PADDING, ACI_RANGE_ALLOC, DB_QUERY
Input Funnel	Application Proxy	DATA, DESTROY, PADDING, DB_UPDATE, ACI_RANGE_ALLOC
Input Funnel	Database engine	PADDING, DB_QUERY, DB_UPDATE, ACI_RANGE_ALLOC
Input Funnel	Input Funnel	CREATE, DATA, DESTROY, PADDING, ACI_RANGE_ALLOC
Onion Router Core	Input Funnel	DATA, DESTROY, PADDING, ACI_RANGE_ALLOC, DB_UPDATE
Application Proxy	Input Funnel	CREATE, DATA, DESTROY, PADDING, DB_QUERY, ACI_RANGE_ALLOC
Database engine	Input Funnel	PADDING, DB_QUERY, DB_UPDATE, ACI_RANGE_ALLOC

Table 5.1.1.3 - Valid Cell Types on Input Funnel Peer Connections

5.1.2 Input Funnel Listener Socket

Input funnels will passively listen for active connections from its peers. This is to enable peers whose relation to a given input funnel is master to actively connect to the core and also enables virtual pipe recovery in the event of a pipe error. The port that the input funnel will listen for connections on will be well known to the input funnel's peers and should probably be a port number that is globally used for pipe connection requests by every network sub-system.

5.1.3 Input Funnel Virtual Pipe Connections

As discussed in section 5.1.1.2, input funnels have a master/slave relationship with each of their peers. During the initialization of an input funnel a virtual pipe connections will be established between the input funnel and each of its peers. The master/slave relationship determines how this connection is established at startup.

On the virtual pipe connections where the input funnel is the master the input funnel will attempt to actively connect to its slave peer by making a connection request to the well known listener socket of the slave and subsequently establishing a new connection. On the virtual pipe connections where the input funnel is the slave the input funnel will passively listen for connection requests from the master peer.

In both cases the input funnel will fork off a child process to handle the pipe initialization so that the key exchange protocol does not cause too much of a disruption of active traffic through the core.

In both cases the master of the pipe connection will initiate the key exchange protocol and will assign the valid ACI range for the virtual pipe.

5.1.4 ACI Range Allocation for Input Funnel Virtual Pipes

In the input funnel sub-system the ACI range allocation must be unique across all virtual pipe connections maintained. This is because the input funnel does not maintain anonymous circuit tables and thus must use the ACI of incoming cells for routing information. If the ACI ranges were not unique routing ambiguities would arise.

The input funnel is responsible for refusing a given ACI range allocation as received from a master peer if it conflicts with a previously assigned range allocation. This should not occur as the input funnel has only a single master peer. Additionally the input funnel must ensure uniqueness of the ACI ranges when assigning those ranges to its slave peers.

5.2 Input Funnel Cell Processing Requirements

An input funnel forwards cells from one virtual pipe to another. Which pipe a given cell is forwarded to depends upon the anonymous circuit that the cell is traveling on, and the type of cell being forwarded. Since the input funnel does not maintain anonymous circuit tables, the rules for forwarding of cells are somewhat simplified in that the forwarding of circuit oriented cells do not depend upon the state or type of the circuit.

The following sections detail the specific cell processing for each valid cell type.

5.2.1 CREATE cells

The input funnel will forward CREATE cells received from application proxies to the onion router core peer. CREATE cells received from any other sub-system constitute and error condition. This error, if it occurs, will be logged as a protocol failure.

5.2.2 DATA cells

The input funnel will forward DATA cells to the appropriate virtual pipe connection as determined by the ACI of the cell. As specified in section 5.1.4 the ACI range for each virtual pipe is unique and thus the mapping from the cells ACI to the virtual pipe it should be forwarded to is unambiguous. If a cell is received on a circuit that does not fall within the ACI range of the virtual pipe the cell will be ignored and dropped and the event will be logged as a protocol failure.

5.2.3 DESTROY cells

The input funnel will forward DESTROY cells to the appropriate virtual pipe connection as determined by the ACI of each cell.

5.2.4 PADDING cells

The input funnel will ignore and drop any received PADDING cells.

5.2.5 DB_QUERY cells

The processing of DB_QUERY cells depends upon the network sub-system that they were received from. DB_QUERY cells can be received from the application proxy or the database engine.

If the DB_QUERY cell is received from the application proxy
then that cell will be forwarded to the database engine.

If the DB_QUERY cell is received from the database engine then the input funnel return a DB_UPDATE cell with a link update for each of it’s virtual pipe connections. These DB_UPDATE cells will be sent on the appropriate circuit as specified in section ??? on the database engine sub-system.

5.2.6 DB_UPDATE cells

The processing of a DB_UPDATE cell depends upon the sub-system it was received from. A DB_UPDATE cell can be received from the database engine, an onion router core, or another input funnel.

5.2.6.1 Processing of a DB_UPDATE Cell Received from the Database Engine

A DB_UPDATE cell received from the database engine will be forwarded onto the virtual pipe addressed by the ACI of the cell. If the ACI of the cell has the value N, then the cell will be forwarded on virtual pipe number N. The ACI of the cell will be set to the ACI of the reserved database cell circuit, i.e. the lowest ACI in the ACI range of the pipe.

5.2.6.2 Processing of a DB_UPDATE Cell Received from an Onion Router Core

A DB_UPDATE cell received from an onion router core will be forwarded to the database engine. It will be forwarded on virtual pipe #0, with a ACI equal to the number of the virtual pipe it was received on.

5.2.6.3 Processing of a DB_UPDATE Cell Received from an Input Funnel

A DB_UPDATE cell received from an input funnel will be forwarded as described in section 5.2.6.1, i.e. identically to a cell received from the database engine.

5.2.7 Input Funnel Error Handling

There are three classes of errors that can occur on an input funnel. These are system errors, virtual pipe errors, and cell errors.

5.2.8 System Error Handling

5.2.9 Virtual Pipe Error Handling

The input funnel will detect all virtual pipe errors as described in section 2.3.8 and implement the recovery mechanisms described in that section.

5.2.10 Cell Error Handling

As described in section 5.1.1.3 only certain types of cells should be received from each of the input funnel's peers and the receipt of an invalid cell type constitutes an error condition. This section details the specific error recovery action that will be taken in the event of receiving an invalid cell on a virtual pipe connection to a given peer.

5.2.10.1 Onion Router Core Cell Errors

Receipt of the following cell types from the onion router core is an error condition:

a) CREATE
b) DB_QUERY
c) DB_UPDATE

In the case of all of the above the errant cell will be ignored and dropped. The event will be logged as a protocol failure.

5.2.10.2 Database Engine Cell Errors

Receipt of the following cell types from the database engine is an error condition:

a) CREATE
b) DATA
c) DESTROY

In the case of all of the above the errant cell will be ignored and dropped. The event will be logged as a protocol failure.

5.2.10.3 Application Proxy Cell Errors

6. Output Funnel Requirements

6.1 Output Funnel Communications Requirements

6.1.1 Output Funnel Peers

Output funnels will maintain virtual pipe connections to neighboring sub-systems known as the output funnel's peers or neighbors.

6.1.1.1 Output Funnel Peer Sub-systems

Output funnel peers will be the following:

a) an output funnel will maintain a single virtual pipe connection to an onion router core.
b) an output funnel must support one or more virtual pipe connections to reply onion processors.
c) an output funnel must support one or more connections to responder proxies.
d) an output funnel may support one connection to an additional output funnel.

6.1.1.2 Master/Slave Relationships Between Output Funnel Peer Sub-systems

As described in section 2.3 each sub-system at one end of a virtual pipe has a master/slave relationship with the sub-system at the other end. The master/slave relationships between the output funnel and its peers are as follows:

a) the output funnel will be the slave to the onion router core
b) the output funnel will be the master to each reply onion processor
c) the output funnel will be the master to each responder proxy.

6.1.1.3 Acceptable Cell Types for Given Peer Connections

The following table lists the acceptable cell types for each of the different virtual pipe connections in an onion routing network.

Sending Sub-System	Receiving Sub-System	Valid Cell Types
Output Funnel	Onion Router Core	DATA, DESTROY, PADDING
Output Funnel	Reply Onion Processor	DATA, DESTROY, PADDING
Output Funnel	Responder Proxy	DATA, DESTROY, PADDING
Onion Router Core	Output Funnel	DATA, DESTROY, PADDING
Reply Onion Processor	Output Funnel	DATA, DESTROY, PADDING
Responder Proxy	Output Funnel	DATA, DESTROY, PADDING

Table 6.1.1.3 - Valid Cell Types on Output Funnel Peer Connections

6.1.2 Output Funnel Listener Socket

Output funnels will passively listen for active connections from its peers. This is to enable peers whose relation to a given output funnel is master to actively connect to the core and also enables virtual pipe recovery in the event of a pipe error. The port that the output funnel will listen for connections on will be well known to the output funnel's peers and should probably be a port number that is globally used for pipe connection requests by every network sub-system.

6.1.3 Output Funnel Virtual Pipe Connections

The virtual pipe connections from the onion router to its peers have less stringent requirements than the usual virtual pipe connection. This section details those requirements that can be eased for the virtual pipe connections to given output funnel peers.

6.1.3.1 Output Funnel to Onion Router Core Virtual Pipe Connection

The virtual pipe connection from the output funnel to the onion router core is simplified in the following ways:

a) Link encryption is optional. This will be specified in the output funnels configuration file.
b) ACI range negotiation is optional. This will be specifed in the output funnels configuration file.
c) The ACI of cells received on this virtual pipe will not be altered when the cell is routed to the appropriate responder proxy.
d) A full anonymous circuit database is not required. The only information that must be maintained about the anonymous circuits on this virtual pipe is:
    a. a mapping from the circuit ACI to a virtual pipe connection to a responder proxy
    b. the state of the circuit, either up or pending, circuits that have no entry in the table are implicitly down.
    c. whether the circuit is a reply circuit
e) No cell buffering is required, all cells can be immediately
f) routed to the appropriate virtual pipe.
g) All cells received from any virtual pipe connection to a responder proxy or a reply onion processor is routed to the virtual pipe connection to the onion router core with the cell ACI left unmodified.
h) All cells received on a reply circuit are routed to the reply onion processor with the cell ACI unmodified.

6.1.3.2 Output Funnel to Responder Proxy Virtual Pipe Connections

The output funnel to responder proxy virtual pipe connections require special attention as they are handled a bit differently than ordinary virtual pipes. This section will discuss the differences between ordinary virtual pipe connections and the virtual pipe connection between the output funnel and the responder proxy.

There will be three groups of responder proxies. The output funnel will be responsible for routing cells to the appropriate responder proxy based on properties of the circuit. The three groups are:

a) Virtual Private Network responder proxy
b) Responder Proxy for long lived connections
c) Responder Proxy for other non-reply circuit connections

6.1.3.2.1 Virtual Private Network Responder Proxy

This responder proxy is used for cells traveling on a virtual private network. This responder is responsible for handling protocol for routing IP packets over the onion routing network which includes the ability to unfragment packets, request retransmissions, etc.

6.1.3.2.2 Long Lived Connection Responder Proxy

This responder proxy is the "first" responder proxy, i.e. a proxy process that will never be terminated. It is intended that long lived connections be routed to this responder so that they don't end up keeping transient responder proxies active longer than they need be. It is possible, on an output funnel with a large amount of long lived connection traffic that this proxy will actually be a group of greater than one proxies.

6.1.3.2.3 General Responder Proxy

This responder proxy handle all other connection traffic. Under light connection loading it is likely that this will be the same proxy as that used for long lived connections, but under heavy connection loading these will be separate, dynamically spawned responders that are active only as long as the connection traffic warrants.

6.1.3.2.4 Output Funnel to Responder Proxy Virtual Pipe Properties

The virtual pipe connections from the output funnel to responder
proxies are simplified in the following ways:

a) No link encryption is needed and thus no key exchange protocol need be performed.
b) No ACI range negotiation is necessary. All ACI's are valid on each of the virtual pipe connections to responder proxies.
c) All cells received on a virtual pipe connection to a responder proxy are routed to the virtual pipe connection to the onion router core with the cell ACI unmodified.
d) No anonymous circuit database is needed for virtual pipe connections to the responder proxy.

6.1.3.2.5 Output Funnel to Responder Proxy Virtual Pipe Management

Unlike ordinary virtual pipe connections, which are more or less permanent connections between peer sub-systems, the virtual pipe connections between the output funnel and the responder proxy are dynamic. They are created and destroyed as needed, dictated by the current connection load.

Further, the output funnel potentially manages several different types of responders. These are the reply onion processor, the virtual private network (VPN) responder, and one or more ordinary responders.

6.1.3.2.5.1 Partitioning of Connections to Different Responder Types

The output funnel will differentiate four types of connections when determining which responder proxy to assign the connection to. These connection types are:

a) ordinary short lived connections
b) ordinary long lived connections
c) VPN connections
d) reply circuit connections

As noted above there are three types of responder proxies. The output
funnel will route connection types to responder types as follows:

a) reply circuit connections are assigned to the reply onion processor
b) VPN connections are assigned to the VPN responder proxy
c) long lived connections are assigned to the permanent responder. This is a responder process that is never retired and is thus always active
d) short lived connections will be assigned to either the permanent responder, if it is the only one active, or it will be assigned to one of the other active ordinary responders in a manner that implements connection load balancing among the active responders.

6.1.3.2.5.2 Spawning of New Responder Processes

The output funnel will maintain a number of responder processes. Some of these will be permanent processes, with a lifetime of that of the output funnel sub-system, and some will be short lived, dynamically spawned processes that are created and destroyed as needed. The responder processes that the output funnel will maintain are as follows:

a) one permanent reply onion processor process
b) one permanent VPN responder proxy process
c) at least one permanent ordinary responder proxy process
d) zero or more transient ordinary responder proxy processes

The output funnel will spawn transient responder proxy processes as needed by the connection load. The output funnel will have a connection threshold function, CT(n), where n is the number of active ordinary responder processes, that will determine when a new responder process must be spawned. If K is the current number of connections, and N is the current number of active ordinary responder processes then:

a) when CT(N) < K < CT(N+1) the output funnel is in normal operation
b) when K > CT(N+1) the output funnel will spawn a new ordinary responder process.
c) when K < CT(N+1) the output funnel will mark one or more responder processes for retirement. This will be discussed further in section 6.1.3.2.5.3.

6.1.3.2.5.3 Retiring Unneeded Responder Processes

The output funnel will attempt to maintain only as many active responder processes as the current connection load requires. Thus the output funnel will implement a scheme for retiring responder processes when the connection load decreases. As in the previous section let K be the current number of connections and N the current number of active ordinary responders. Then the responder retirement scheme is as follows:

a) when K < CT(M) then N-M+1 responders will be tagged for retirement.
b) the output funnel will not assign any further connections to a responder that has been tagged for retirement.
c) if the connection load increases so that K > CT(M), then a responder from the list of tagged responders will be untagged.
d) when the number of connections on a responder tagged for retirement reaches zero, that responder will be closed.
e) if it is determined that a responder tagged for retirement has remained open longer than a chosen threshold, due to one or more long lived connections, then that responder will be untagged and a different responder will be chosen for retirement.
f) a least recently used ordering will be maintained among all of the transient ordinary responders. When a responder is untagged it will be placed at the bottom of the order. The responder chosen to be tagged in it's place, if necessary, will be chosen from the top of the order.
g) When a new responder proxy process is spawned it will be placed at the top of the order so as to implement a pseudo LIFO.

6.1.3.3 Output Funnel to Reply Onion Processor Virtual Pipe Connections

All reply circuits are assigned to a reply onion processor which handles the final connection to the anonymous connection acceptor. The output funnel will maintain one or more connections with reply onion processors. These connections to reply onion processors are handled similarly to the handling of responder proxies. There are three types reply onion processors:

a) Permanent Reply Onion Processors
b) Specific Dynamic Reply Onion Processors
c) Generic Dynamic Reply Onion Processor

The virtual pipe connection between the output funnel and the reply onion processor has the same properties as the virtual pipe connection between the output funnel and the responder proxies. Namely:

a) Lnk encryption is optional. This will be specified in the output funnel’s configuration file.
b) No ACI range negotiation is necessary. All ACI's are valid on each of the virtual pipe connections to responder proxies.
c) All cells received on a virtual pipe connection to a reply onion processor are routed to the virtual pipe connection to the onion router core with the cell ACI unmodified.
d) No anonymous circuit database is needed for virtual pipe connections to the reply onion processor.

6.1.3.3.1 Permanent Reply Onion Processors

Permanent reply onion processors are those for which the connection between them and the output funnel is opened at initialization time and kept up for the lifetime of the output funnel sub-system. These permanent reply onion processors will be listed in the output funnel's configuration file. Permanent reply onion processors are intended to service heavily used anonymous connection acceptors and will generally have some sort of administrative tie to the organization running the output funnel. For example if an organization running an output funnel also desires to provide anonymous content via an anonymous web server then that organization will most likely also run a reply onion processor and the connection between that processor and the output funnel will be permanent.

6.1.3.3.2 Specific Dynamic Reply Onion Processors

Dynamic reply onion processors are spawned and retired as dictated by specific connection requests. As noted in section 2.4 on onions reply onions can specify what reply onion processor should be used to service the circuit. A reply onion processor used to service a specific connection acceptor or group of connection acceptors is known as a specific reply onion processor. Thus reply onion processors which service specific connection acceptors and are spawned and retired as needed are referred to as specific dynamic reply onion processors. The output funnel will spawn a child process to manage the dynamic reply onion processors in a very similar fashion to the management of responder proxies. One difference is that which reply onion processor to use is specified in the reply onion. The output funnel simply has to parse the reply onion processor host name or address and port out of the reply onion and assign the circuit to that reply onion processor. No load balancing will be attempted on specific dynamic reply onion processors.

6.1.3.3.3 Generic Dynamic Reply Onion Processor

Generic dynamic reply onion processors are those processors which will be spawned and retired as needed and who are not dedicated to servicing any particular group of anonymous connection acceptors. Reply circuits whose reply onions did not specify a particular reply onion process will be assigned to the generic dynamic reply onion processor. If there are currently more than one generic dynamic reply onion processors active then an attempt will be made by the output funnel to balance the connection load among them. Generic dynamic reply onion processors will be spawned and retired according to the same rules as the generic responder proxies.

6.2 Output Funnel Cell Processing Requirements

This section details the processing of cells received at the output funnel, and further details the processing of data received in raw format from the connection acceptors.

6.2.1 Circuit Oriented Cells

As discussed in section 2.2.4 anonymous circuits can be in one of five states. The anonymous circuits maintained on the output funnel are somewhat different than those maintained on onion router cores in that the anonymous circuit states on an output funnel don't correspond exactly to the same states on an onion router core.

6.2.1.1 Anonymous circuit States on an Output Funnel

The states of a anonymous circuit as discussed in section 2.2.4 are interpreted slightly differently on an output funnel. The table below details the states of a anonymous circuit on an output funnel and describes the events triggering a state transition.

Circuit State	Description

Circuit State	Description
Down	The down state is an implicit state that begins with the freeing of the circuit data structures after receipt of a DESTROY acknowledgment and ends with the receipt of the first DATA cell on the circuit. From the down state the circuit enters the up state.
Up	The up state is an explicit state that begins with the receipt of the first DATA cell on the circuit and ends after either the sending or receipt of a DESTROY cell. From the up state the circuit enters the pending state.
Pending	The pending state is an explicit state that begins after the receipt or sending of a DESTROY cell and ends with the receipt or sending of a DESTROY acknowledgment. From the pending state the circuit enters the implicit down state.

Table 6.2.1.1 - Anonymous circuit States on an Output Funnel

6.2.1.2 DATA Cells

6.2.1.2.1 Processing of DATA Cells in the Down State

Upon receipt of a DATA cell on a circuit in the down state the output funnel will create the necessary forward and reverse circuit structures and mark the circuit as created.

The contents of the DATA cell, which is the standard structure, will be inspected in order to determine if the new circuit is a reply circuit. The cell will then be routed to an appropriate responder proxy or, if the circuit is a reply circuit, the reply onion processor.

6.2.1.2.2 Processing of DATA Cells in the Up State

Upon receipt of a DATA cell on a circuit in the created state the output funnel will route that cell to the appropriate responder proxy.

6.2.1.2.3 Processing of DATA Cells in the Pending State

Upon receipt of a DATA cell on a circuit in the pending state the output funnel will ignore and drop that cell.

6.2.1.3 DESTROY Cells

6.2.1.3.1 Processing of DESTROY Cells in the Down State

The receipt of a DESTROY cell on a circuit in the down state is an error condition. The cell will be ignored and dropped.

6.2.1.3.2 Processing of DESTROY Cells in the Up State

Upon receipt of a DESTROY cell on a circuit in the up state, the output funnel will send the DESTROY cell back along the anonymous circuit it was received on and mark that circuit as pending.

6.2.1.3.3 Processing of DESTROY Cells in the Pending State

Upon receipt of a DESTROY cell on a circuit in the pending state the output funnel will free any data structures used to manage that circuit and the circuit will implicitly be marked down.

6.2.2 Pipe Oriented Cells

The only pipe oriented cell received by the output funnel is the PADDING cell which is ignored and dropped.

6.3 Output Funnel Error Handling Requirements

6.3.1 Local System Error Handling

6.3.2 Anonymous circuit Error Handling

6.3.3 Virtual Pipe Error Handling

7. Cryptographic Processor Requirements

7.1 Cryptographic Processor Communications Requirements

7.1.1 Cryptographic Processor Peers

The cryptographic processor sub-system has a single peer, the onion router core. The cryptographic processor is the slave to the onion router core on the virtual pipe connection between the two sub-systems.

The acceptable cell types traveling on the virtual pipe connection between the cryptographic processor and the onion router core are as follows:

Sending Sub-System	Receiving Sub-System	Valid Cell Types
Cryptographic Processor	Onion Router Core	CREATE, DESTROY, ONION_INFO, PADDING
Onion Router Core	Cryptographic Processor	CREATE, DESTROY, PADDING

Table 7.1.1 - Valid Cell Types on the Cryptographic Processor Peer Connection

7.1.2 Cryptographic Processor Listener Socket

The cryptographic processor will listen for connection requests from the onion router core on a port well known by the onion router core. The cryptographic processor will passively connect to the onion router core via accepting connections received on this listener socket.

7.2 Cryptographic Processor Cell Processing Requirements

This section details the cell processing requirements for the cryptographic processor sub-system. The cryptographic processor does not distinguish between the types of circuits it receives
cells on. It treats the cells it receives identically for each of the various types of circuits discussed in section 2.2.5.

7.2.1 Circuit Oriented Cells

The cryptographic processor has a very rudimentary understanding of circuits.

7.2.1.1 Anonymous circuit States on the Cryptographic Processor

The states of a anonymous circuit as discussed in section 2.2.4 are interpreted differently on the cryptographic processor. The table below details the states of a anonymous circuit on the cryptographic processor and describes the events triggering a state transition.

Anonymous circuit State	Description
Down	The down state is an implicit state that begins with the receipt of a DESTROY ACK from the onion router core or begins with the forwarding of the last processed CREATE cell back to the onion router core, or begins with the forwarding of the ONION_INFO cell back to the onion router core when that core is the last hop of the circuit (i.e. no CREATE cells need to be forwarded on). The down state ends with the receipt of the first CREATE cell on the circuit. From the down state the circuit enters the created state.
Created	The created state is an explicit state that begins with the receipt of the first CREATE cell on this circuit and ends with the forwarding of the ONION_INFO cell back to the onion router core. From the created state the circuit enters either the up state, if the onion processing was successful, or the pending state if the received onion was invalid or if a DESTROY cell was received prior to receiving all of the CREATE cells containing the onion.
Up	The up state is an explicit state that begins with the forwarding of the ONION_INFO cell back to the onion router core and ends with either the forwarding of the last processed CREATE cell back to the onion router core or ends with the receipt of a DESTROY cell from the onion router core. From the up state the circuit either implicitly enters the down state, if the forwarding of the CREATE cells back to the onion router core was not interrupted by the receipt of a DESTROY cell, and enters the pending state if a DESTROY cell was received prior to forwarding the processed CREATE cells.
Pending	The pending state is an explicit state that begins with either the receipt of a DESTROY cell from the onion router core or with the sending of a DESTROY cell to the onion router core in the event of an invalid onion or other error rendering the circuit invalid. From the pending state the circuit implicitly enters the down state.

Table 7.2.1.1 - Anonymous circuit States on a Cryptographic Processor

7.2.1.2 CREATE Cells

7.2.1.2.1 Processing of CREATE Cells in the Down State

Upon receipt of a CREATE cell on a circuit in the down state the cryptographic processor will allocate a data structure for managing that circuit, add that cell to the head of a cell list maintained for that circuit, and set the count of such cells to one. It will then mark the circuit in the created state.

7.2.1.2.2 Processing of CREATE Cells in the Created State

Upon receipt of a CREATE cell on a circuit in the created state the cryptographic processor will add that cell to the end of the cell list for the circuit and increment the count of such cells.

If it is determined that the number of cells that has been currently received is equal to the number of cells that comprise a complete onion then those cells will be processed in order to obtain the information in the top layer of the onion. That processing consists of the following:

a) the payloads of all of the buffered CREATE cells are catenated to a single buffer. That buffer thus contains the full onion ready to be decrypted.
b) the onion is decrypted
c) the first bit in the onion is verified to be a zero
d) the following data fields are parsed out of the top onion layer:
    a. version
    b. flags
    c. backward crypto function index
    d. forward crypto function index
    e. global identifier of next hop
    f. expiration time of the onion
    g. the key seed material

These data are placed into the formatted payload of an ONION_INFO cell

e) the entire onion is digested with SHA and that digest is placed into the payload of the ONION_INFO cell
f) that ONION_INFO cell is forwarded back to the onion router core on the same anonymous circuit it was received on
g) the circuit state is marked as up
h) the contents of the onion buffer is adjusted such that the bottom N-1 layers (N total layers in an onion) are moved to the top of the buffer, overwriting the top layer, and the bottom layer of the onion buffer is filled with random bits.
i) the onion buffer is repacked into CREATE cells and those cells are forwarded to the onion router core on the circuit that the original CREATE cells were received on
j) the data structures used to manage the circuit are cleaned up and freed
k) the circuit implicitly enters the down state

7.2.1.2.3 Processing of CREATE Cells in the Up State

Once a circuit has reached the up state on the cryptographic processor all of the CREATE cells for that circuit should have been received. Thus receipt of any further CREATE cells is an error condition. The cryptographic processor will send a DESTROY cell back to the onion router core on this circuit and mark the circuit state as pending. This event will be logged as a protocol failure.

7.2.1.2.4 Processing of CREATE Cells in the Pending State

Once a circuit has entered the pending state any CREATE cells received on that circuit will be ignored and dropped. The event will be logged as a protocol failure.

7.2.1.3 DESTROY Cells

7.2.1.3.1 Processing of DESTROY Cells in the Down State

DESTROY cells received on a circuit in the down state will be ignored and dropped.

7.2.1.3.2 Processing of DESTROY Cells in the Created State

Upon the receipt of a DESTROY cell on a circuit in the created state the cryptographic processor will do the following:

a) echo the DESTROY cell back to the onion router core on the same circuit that it was received on
b) clean up and free any data structures used to manage the circuit including freeing any CREATE cells that are being buffered
c) implicitly mark the circuit state as down

7.2.1.3.3 Processing of DESTROY Cells in the Up State

The processing of a DESTROY cell on a circuit in the up state is identical to that done when a DESTROY cell is received on a circuit in the created state.

7.2.1.3.4 Processing of DESTROY Cells in the Pending State

Upon receipt of a DESTROY cell on a circuit in the pending state the cryptographic processor will clean up and free any data structures used to manage the circuit and the circuit state will be implicitly marked as down.

7.2.2 Pipe Oriented Cells

The only pipe oriented cell received by the cryptographic processor is the PADDING cell which is ignored and dropped.

7.3 Cryptographic Processor Error Handling

8. Responder Proxy Requirements

This section details the requirements for the responder proxy sub-system.

8.1 Responder Proxy Communications Requirements

This section details the communications requirement for the responder proxy sub-system.

8.1.1 Responder Proxy Peers

This section details information regarding the peer sub-systems of the responder proxy.

8.1.1.1 Responder Proxy Peer Sub-Systems

The responder proxy has one peer and that is the output funnel sub-system.

8.1.1.2 Master/Slave Relationships Between Responder Proxy Peer Sub-Systems

The responder proxy is slave to its output funnel peer.

8.1.1.3 Acceptable Cell Types for Given Peer Connections

The acceptable cell types for the output funnel to responder proxy connection are:

Sending Sub-System	Receiving Sub-System	Valid Cell Types
Responder Proxy	Output Funnel	DATA, DESTROY, PADDING
Output Funnel	Responder Proxy	DATA, DESTROY, PADDING

Table 8.1.1.3 - Valid Cell Types on Responder Proxy Peer Connections

8.1.2 Responder Proxy Listener Socket

The responder proxy will listen for an incoming connection request from its peer output funnel on a well known listener port. This port will be known to its peer output funnel.

8.1.3 Responder Proxy to Connection Acceptor Socket Connections

The responder proxy is responsible for making the ultimate connection to the connection acceptor. In order to do this the responder proxy must interpret the standard structure and destination host info and take steps to obtain a valid IP address and port for the connection acceptor based on that data. Then the responder proxy must actively make the connection to the destination host.

8.1.3.1 Destination Host Address Formats

The responder proxy must interpret three support destination host address formats:

a) ASCII string - Fully qualified domain name & port number
b) ASCII string - Dotted decimal IP address & port number
c) ASCII string - Fully qualified domain name of domain name server of host desired to be connect to. The specified host is expected to be able to respond to a DNS query for MX records of the connection acceptor.

8.1.3.2 Reserved Port Requirement for Certain Applications

Some applications, most notably the Berkeley "R" command applications such as rlogin require that a client connection request come from a reserved port. In order to obtain a socket descriptor for a reserved port the system call rresvport(3N) must be called. The responder proxy must be able to recognize that a given connection will require that the connection request come from a reserved port and use the rresvport(3N) call rather than the usual socket(3N) call in these cases.

8.1.3.3 Connection Retries

The responder proxy may be directed to make multiple attempts to connect to the destination host. The number of attempts that should be made by the responder proxy is contained in the "retry count" field of the standard structure as described in section 2.4. The responder proxy will repeat active connection attempts until either a successful connection to the destination host is made or until the number of retries reaches the specified retry count.

8.1.3.4 Acknowledgment Upon Connection to Destination Host

The responder proxy will send a one byte acknowledgment after attempting to connect to the destination host. If the connection attempt was successful then the acknowledgment will contain a value specifying no error. If the connection attempt was not successful then the negative acknowledgment will contain information as to why the connection was unsuccessful. The acknowledgment is sent as a DATA cell with a one byte payload.

If the connection attempt was unsuccessful the responder proxy will send a DESTROY cell back down the anonymous circuit to take down that circuit.

8.1.3.5 Circuit Termination

The responder proxy must detect when the session with the connection acceptor is completed and terminate the circuit accordingly. The way in which the responder detects this is by reading and EOF on the connection to the connection acceptor. This signifies that the connection acceptor has closed the connection and thus no more data will be arriving on that connection.

When the responder proxy detects the EOF condition on the connection to the connection acceptor it will send a DESTROY cell to the output funnel on the anonymous circuit being used for this connection. It will then mark the circuit as pending and wait for receipt of a DESTROY acknowledgment upon which it will free any circuit data structures and the circuit will be marked implicitly down.

8.2 Responder Proxy Cell Processing Requirements

This section details the cell processing requirements for each type of cell received by the responder proxy. The responder proxy is only used for non-reply circuits, and tunneled circuits are transparent to the responder proxy.

8.2.1 Circuit Oriented Cells

The responder proxy must process the DATA and DESTROY circuit oriented cell types.

8.2.1.1 Anonymous circuit States on a Responder Proxy

The states of a virtual as discussed in section 2.2.4 are interpreted slightly differently on the responder proxy. The table below details the states of a anonymous circuit on a responder proxy and describes the events triggering a state transition.

Anonymous circuit State	Description
Down	The down state is an implicit state that begins with the freeing of the circuit data structures after the receipt of a DESTROY acknowledgment and ends with the receipt of the first DATA cell on the circuit. From the down state the circuit enters the up created state.
Created	The created state is an explicit state that begins with receipt of the first DATA cell on the circuit and ends after the one byte ACK or NAK has been sent back toward the connection initiator. From the created state the circuit either enters the up state, if a successful connection to the connection acceptor was made, or the pending state, if the connection attempt was unsuccessful or some other error occurred.

Anonymous circuit State	Description
Up	The up state is an explicit state that begins after sending the one byte ACK back to the connection initiator and ends either upon receipt of a DESTROY cell from the output funnel, or upon detection of an EOF on the connection acceptor socket. From the up state the circuit enters the pending state.
Pending	The pending state is an explicit state that begins after either the receipt of a DESTROY cell from the output funnel or upon the sending of a DESTROY cell to the output funnel after detecting an EOF on the connection acceptor socket. The pending state ends upon an orderly shutdown of the connection acceptor socket and freeing of the circuit data structures in the first case, and upon the receipt of a DESTROY ACK, and freeing of the circuit data structures in the second case. In either case the circuit enters the down state from the pending state.

Table 8.2.1.1 - Anonymous circuit States on a Responder Proxy

8.2.1.2 DATA Cells

8.2.1.2.1 Processing of DATA Cells in the Down State

Upon receipt of a DATA cell on a circuit in the down state the responder proxy will interpret the cell as the standard structure of a new circuit, parse the data in that structure out and store it for later use, allocate new circuit data structures as needed, and mark the circuit as created.

If the standard structure is invalid, the responder proxy will send a DESTROY cell back along the circuit towards the initiator and will maintain the circuit in the down state.

8.2.1.2.2 Processing of DATA Cells in the Created State

Upon entering the created state the responder will begin buffering DATA cells until it is able to either successfully parse out a host address and port number, whose formats were specified in the standard structure, or determine that the received host address and/or port number is invalid.

In the first case it will convert the received host address and port number as determined by the address format specified in the standard structure and will attempt to connect to this host.

If the connection is successful then a one byte ACK is sent in a DATA cell, back along the circuit towards the initiator, and the circuit is marked as up.

If the connection is unsuccessful, or if there were some error in the host address or port number info, then a one byte NAK will be sent back along the circuit towards the initiator immediately followed by a DESTROY cell. In this case the circuit will be marked as pending.

8.2.1.2.3 Processing of DATA Cells in the Up State

Upon receipt of a DATA Cell on a circuit in the up state the responder will write the payload of that DATA cell to the appropriate connection acceptor socket as determined by the ACI of the DATA cell.

8.2.1.2.4 Processing of DATA Cells in the Pending State

DATA cells received on a circuit in the pending state will be ignored and dropped.

8.2.1.3 DESTROY Cells

8.2.1.3.1 Processing of a DESTROY Cell in the Down State

A DESTROY cell received on a circuit in the down state will be ignored and dropped.

8.2.1.3.2 Processing of a DESTROY Cell in the Created State

Upon receipt of a DESTROY cell on a circuit in the created state the responder proxy will perform the following steps:

a) echo the received DESTROY cell back along the circuit towards the initiator
b) clean up and free any data structures used to manage this circuit
c) the circuit will implicitly enter the down state

8.2.1.3.3 Processing of a DESTROY Cell in the Up State

Upon receipt of a DESTROY cell on a circuit in the up state the responder proxy will perform the following steps:

a) close the socket to the connection acceptor in an orderly fashion discarding any pending data on that socket.
b) send a DESTROY cell back along the circuit towards the initiator
c) clean up and free any data structures used to manage the circuit
d) the circuit implicitly enters the down state

8.2.1.3.4 Processing of a DESTROY Cell in the Pending State

Upon receipt of a DESTROY cell on a circuit in the pending state the responder proxy will clean up and free and data structures used to manage the circuit and the circuit will implicitly enter the down state.

8.2.2 Pipe Oriented Cells

The only pipe oriented cell that the responder proxy will accept is the PADDING cell which will be ignored and dropped.

8.3 Responder Proxy Error Handling Requirements

The responder proxy postcrypts the cells for the return trip back along the reverse route to the session/connection initiator, so it basically performs the same cell encrytion related tasks as the application proxy except in reverse.

8.3.1 Local System Error Handling

8.3.2 Anonymous circuit Error Handling

8.3.3 Virtual Pipe Error Handling

9. Reply Onion Processor Requirements

This section details the requirements for the reply onion processor sub-system. The reply onion processor handles ultimate destination connections for reply circuits.

9.1 Reply Onion Processor Communications Requirements

9.1.1 Reply Onion Processor Peers

The reply onion processor sub-system has a single peer sub-system, the output funnel. The reply onion processor is slave to the output funnel peer on the virtual pipe connection between those two sub-systems. The table below lists the acceptable cell types on the virtual pipe connection between the reply onion processor and the output funnel.

Sending Sub-System	Receiving Sub-System	Valid Cell Types
Reply Onion Processor	Output Funnel	DATA, DESTROY, PADDING
Output Funnel	Reply Onion Processor	DATA, DESTROY, PADDING

Table 9.1.1 - Valid Cell Types on Reply Onion Processor Peer Connections

9.1.2 Reply Onion Processor to Connection Acceptor Socket Connections

The reply onion processor is additionally responsible for making socket connections to anonymous connection acceptors and participating in the appropriate application protocol. The reply onion processor is responsible for actively making the connection to the connection acceptor and for managing any protocol setup required before forwarding user application data to the connection acceptor. The specific setup required by each supported application protocol will be detailed in the following sections.

9.1.2.1 Generic Reply Circuit Processing

Every reply circuit requires some generic processing by the reply onion processor. This processing involves the following:

a) Receiving and parsing the reply onion processor header
b) Parsing the destination host address and port info
c) Parsing any application specific data
d) Establishment of the connection with the destination host
e) Sending one byte ACK/NAK back towards the connection initiator.
f) Initialization of the application protocol.

Step (f) will be discussed in further detail for each application protocol support in the following sections.

9.1.2.2 Raw Socket Application Protocol

9.1.2.3 HTTP Application Protocol

9.1.2.4 SMTP Application Protocol

9.1.2.5 Rlogin Application Protocol

9.2 Reply Onion Processor Cell Processing Requirements

9.3 Reply Onion Processor Error Handling

10. Database Engine Requirements

10.1 Database Engine Communications Requirements

10.1.1 Database Engine Peers

Database Engines will maintain virtual pipe connections to neighboring sub-systems known as the database engine's peers or neighbors. A few definitions are in order.

Primary Peer

> The database engine's primary peer is the sub-system to which it communicates with directly via a virtual pipe. For example if a database engine maintains a virtual pipe connection to an onion router core then that onion router core is the database engine's primary peer.

Neighboring Peer

> A neighboring peer of a database engine is the a sub-system that is directly connected to the primary peer of the database engine.

10.1.1.1 Database Engine Peer Sub-Systems

Database engines always have only a single primary peer. The sub-systems which can be the primary peer of a database engine are as follows.

a) Onion Router Core
b) Input Funnel
c) Application Proxy

10.1.1.2 Master/Slave Relationships Between Database Engine Peer Sub-Systems

The database engine is slave to each of its peer sub-systems.

10.1.1.3 Acceptable Cell Types for Given Peer Connections

10.1.1.3.1 Acceptable Cell Types

The acceptable cell types for virtual pipe connections between the database engine and its peers are as follows.

Sending Sub-System	Receiving Sub-System	Valid Cell Types
Database Engine	Onion Router Core	DB_QUERY, DB_UPDATE, PADDING
Database Engine	Input Funnel	DB_QUERY, DB_UPDATE, PADDING
Database Engine	Application Proxy	DB_QUERY, DB_UPDATE, PADDING
Onion Router Core	Database Engine	DB_QUERY, DB_UPDATE, PADDING
Input Funnel	Database Engine	DB_QUERY, DB_UPDATE, PADDING
Application Proxy	Database Engine	DB_QUERY, DB_UPDATE, PADDING

Table 10.1.1.3.1 - Valid Cell Types on Database Engine Virtual Pipe Connections

10.1.1.3.2 Acceptable Cell Sub-types

Since all database cell types are valid on all the virtual pipe connections between database engines and their peers it is necessary to refine the specification by database cell sub-type. In order to do this it is necessary to divide logical connections into two groups. The first group consists of queries and updates from a database engine directly to its primary peer sub-system. The second group consists of queries and updates from one database engine to another database engine where the second engine has a primary peer that is a neighboring peer of the first database engine.

10.1.1.3.2.1 Database Engine to Primary Peer Logical Connections

Database engine to primary peer logical connections consist of those connections where data is exchanged between a database engine and its primary peer. The tables below details the valid database cell sub-types for each connection.

Sending Sub-System	Receiving Sub-System	Local Topology Query	Global Topology Query	Specific Topology Query	Local Key Query	Global Keys Query	Specific Key Query
Database Engine	Onion Router Core	YES	NO	NO	YES	NO	NO
Database Engine	Input Funnel	YES	NO	NO	NO	NO	NO
Database Engine	Application Proxy	YES	NO	NO	NO	NO	NO
Onion Router Core	Database Engine	NO	NO	NO	NO	NO	NO
Input Funnel	Database Engine	NO	NO	NO	NO	NO	NO
Application Proxy	Database Engine	NO	NO	NO	NO	NO	NO

Table 10.1.1.3.2.1A - Valid Database Query Cell Sub-Types for Various Database Primary Peer Connections

Sending Sub-System	Receiving Sub-System	Local Topology Update	Global Topology Update	Specific Topology Update	Local Key Update	Global Keys Update	Specific Key Update
Database Engine	Onion Router Core	NO	NO	NO	NO	NO	NO
Database Engine	Input Funnel	NO	NO	NO	NO	NO	NO
Database Engine	Application Proxy	NO	NO	NO	NO	NO	NO
Onion Router Core	Database Engine	YES	NO	NO	YES	NO	NO
Input Funnel	Database Engine	YES	NO	NO	NO	NO	NO
Application Proxy	Database Engine	YES	NO	NO	NO	NO	NO

Table 10.1.1.3.2.1B - Valid Database Update Cell Sub-Types for Various Database Primary Peer Connections

10.1.1.3.2.2 Database Engine to Neighboring Peer Database Engine Logical Connections

Database engine to database engine logical connections occur when a given database engine either requests or updates a second database engine where the primary peer of the second database engine is a neighboring peer of the first. The two network sub-systems lying on the path between the database engines simply forward the cells unaltered between the two database engines. The tables below detail the valid database cell sub-types for each type of logical connection. Note that the tables below specify the valid cell sub-types on a per leg basis. It is to be understood that the database engines comprise the endpoints of database to database connections.

Sending Sub-System	Receiving Sub-System	Local Topology Query	Global Topology Query	Specific Topology Query	Local Key Query	Global Keys Query	Specific Key Query
Database Engine	Onion Router Core	NO	YES	YES	NO	YES	YES
Database Engine	Input Funnel	NO	YES	YES	NO	YES	YES
Database Engine	Application Proxy	NO	YES	YES	NO	YES	YES
Onion Router Core	Database Engine	NO	YES	YES	NO	YES	YES
Input Funnel	Database Engine	NO	YES	YES	NO	YES	YES
Application Proxy	Database Engine	NO	YES	YES	NO	YES	YES

Table 10.1.1.3.2.2A - Valid Database Query Cell Sub-Types for Various Database to Database Logical Connections

Sending Sub-System	Receiving Sub-System	Local Topology Update	Global Topology Update	Specific Topology Update	Local Key Update	Global Keys Update	Specific Key Update
Database Engine	Onion Router Core	NO	YES	YES	NO	YES	YES
Database Engine	Input Funnel	NO	YES	YES	NO	YES	YES
Database Engine	Application Proxy	NO	YES	YES	NO	YES	YES
Onion Router Core	Database Engine	NO	YES	YES	NO	YES	YES
Input Funnel	Database Engine	NO	YES	YES	NO	YES	YES
Application Proxy	Database Engine	NO	YES	YES	NO	YES	YES

Table 10.1.1.3.2.2B- Valid Database Update Cell Sub-Types for Various Database to Database Logical Connections

10.1.2 Database Engine Listener Socket

The database engine sub-system will maintain an open socket upon which it will listen for incoming connection requests from its primary peer. This is to facilitate virtual pipe recovery in the event that the primary peer sub-system goes down. In the event that the primary peer goes down it will attempt to actively connect to the database engine on this listener socket.

10.1.3 Database Engine Virtual Pipe Connections

The database engine maintains a single virtual pipe connection to its primary peer. That virtual pipe has the following properties based upon whether the database engine is co-hosted with its primary peer on the same machine or whether they are hosted on separate machines.

10.1.3.1 Database Engine Virtual Pipe Properties when Co-hosted with its Primary Peer

When a database engine and its primary peer are hosted on the same machine the virtual pipe connecting them has the following properties.

a) the pipe will not be encrypted
b) the pipe will connect using a Unix pipe
c) the ACI range is not defined, no range negotiation will take place

10.1.3.2 Database Engine Virtual Pipe Properties when its Primary Peer runs on a Separate Machine

When a database engine and its primary peer are hosted on separate machines the virtual pipe connecting them has the following properties.

a) the pipe will be link encrypted
b) the pipe will connect using a Berkeley socket
c) the ACI range is not defined, no range negotiation will take place

10.1.3.3 Anonymous circuit Assignment

Anonymous circuits residing on the virtual pipe connecting the database engine with its primary peer are assigned based on the numbering of the virtual pipes of the primary peer. Topology updates from a primary peer to its database engine concerning the link state of virtual pipe N (as numbered by the primary peer), will be sent to the database engine on anonymous circuit N. ACI number zero is reserved for use by an onion router core (if one is connected to the given database engine) for responding to local key certificate queries sent by the database engine. If the database engine is attached to an input funnel or an application proxy then anonymous circuit number zero remains unused.

Note that the primary peer of a database engine does not report link state information about its peers that do not act as primary peers to another database engine. I.e. a primary peer will only report link state information about onion router cores, input funnels, and application proxies.

An example of a typical configuration should help to clarify this. Suppose we have the following setup.

Onion router core (ORC) #1 has the following peers on the following virtual pipe numbers.

Onion Router Core Peer Sub-System	Virtual Pipe Number on ORC #1	Anonymous circuit used by ORC #1 to report link status of this peer (on virtual pipe #0)
Database Engine	0	N/A
ORC #2	1	1
ORC #3	2	2
Application Proxy	3	3
Input Funnel	4	4
Output Funnel	5	N/A
Crypto Processor	6	N/A

Table 10.1.3.3 - Database Engine Anonymous Circuit Example Configuration

10.1.3.4 Database Engine Virtual Pipe Numbering on the Primary Peer

From the point of view of the primary peer this virtual pipe will be the “zeroth” pipe, i.e. in the array of virtual pipes that the primary peer maintains the pipe to the database engine will be the first element in that array and thus will occupy the “zeroth” array slot.

10.2 Database Engine Database Requirements

10.2.1 Topology Database

Each DBE has a topology database, and it contains the topology of the entire OR network. More specifically, it contains the link states of all pipes connected to OR cores. This database is updated whenever a pipe comes up because the OR cores on both sides of the pipe send updates when a pipe comes up. Additionally, OR cores send updates when they detect that one of their pipes are down (not in the up state).

10.2.2 Key Database

Each DBE has a key database, ant it contains the public keys for all of the pipes link encryption. The primary peer DBE for an application proxy and output funnel are the only sub-systems which require all of the pipe link encryption keys because they are the only sub-systems which pre/post encrypt onions which traverse the OR network. The other sub-systems (OR cores) only need the pipe link keys for the pipes which they maintain. However, since there is no differentiation between DBEs, all DBEs have all of the pipe link encryption keys.

10.2.3 Access List Database

Each DBE has an access list (ACL) database, ant it contains the ACL for the entire OR network. The primary peer DBE for an application proxy and output funnel are the only sub-systems which require the ACL information because they are the only sub-systems which accept connections into the OR network. The other sub-systems (OR cores) only don't need the ACL information. However, since there is no differentiation between DBEs, all DBEs have ACL databases.

10.3 Database Engine Cell Processing Requirements

10.3.1 Processing of DB_UPDATE Cells

The database update are directed to particular GIDs which are only assigned to OR cores. Updates are really directed to the primary peer DBE of the OR core designated by the GID. Therefore, DBEs are associated with the GID of their primary peer OR core. The primary peer OR core checks the destination GID, and if the update is destined for that GID the query is passed to the cores associated DBE. The DBE will accept signed & unsigned updates originating from its primary peer core. However, updates originating from other sub-systems must have valid signatures for the DBE to process them. If the update is for this DBE it is processed by the DBE otherwise it tries to forward the update along its path. The update is forwarded along its path as long as the update is signed; otherwise, it is dropped. The validity of the signature of updates to be forwarded aren't verified by this DBE. They will be verified by the destination DBE. This DBE signs updates from its primary peer core if they are destined for a different DBE. The update is dropped if this DBE has seen it already.

10.3.2 Processing of DB_QUERY Cells

The database queries are directed to particular GIDs which are only assigned to OR cores. Queries are really directed to the primary peer DBE of the OR core designated by the GID. Therefore, DBEs are associated with the GID of their primary peer OR core. The primary peer OR core checks the destination GID, and if the query is destined for that GID the query is passed to the cores associated DBE. The DBE extracts the requested data from the database hash table and generates as many DB_UPDATE cells as necessary to incorporate all of the requested data. The destination GID in the DB_UPDATE is the GID making the query.

10.3.3 Processing of PADDING Cells

Padding cells are ignored and dropped by the database engine.

10.4 Database Engine Error Handling Requirements