CBMEN ENCODERS

Content-Based Mobile Edge Networking Program

Edge Networking with Content-Oriented Declarative Enhanced Routing and Storage

Contract Number: N66001-12-C-4051

ENCODERS Configuration Manual

Version 2.0 – *September 2, 2014*

Submitted to:

Defense Advanced Research Projects Agency (*DARPA*), Strategic Technology Office (*STO*)
Attention: Dr. Wayne Phoel, Email: wayne.phoel@darpa.mil

Submitted by:

SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025 USA
Attention: Dr. Carolyn Talcott, Phone: (650) 859-3044, Email: clt@csl.sri.com

Table of Contents

1 Scope

The purpose of this document is to describe the configuration options for SRI's ENCODERS (Edge Networking with Content-Oriented Declarative Enhanced Routing and Storage) software that was developed for DARPA's *Content-Based Mobile Edge Networking (CBMEN)* Program. This document is intended for system administrators who are setting up an ENCODERS network. SRI's Build and Deployment Manual should be used to install ENCODERS on the network nodes before configuration.

With this manual, ENCODERS can be configured to tune the performance of the network to the application. The types of content being transferred in an application and the connectivity of the network (among other things) may influence the configuration selected. Another manual, SRI's Technical Documentation is intended for developers who may be enhancing or modifying the system. SRI's Final Reports for Phases 1 and 2 and the ENCODERS Software Design Description give a high-level introduction of the architecture and the ENCODERS functionality and contain the results of our performance evaluations. Figure 1 show the architecture figure, which depicts the Managers mentioned in this document.

ENCODERS is SRI's CBMEN content-based networking solution. It builds on and extends both our DTN work and Haggle (see the Software Design Description), an existing modular, open-source, content-based mobile networking framework with a simple but expressive attribute-based mechanism for describing content, expressing interest, and matching content to interest. SRI started with the Haggle open-source code base and made major improvements in it, including improving performance and extensions for utility-based cache management, semantic annotations and security.

Based on the Haggle architecture, ENCODERS is event-driven, modular, and layer-less, which provides flexibility and scalability. Central in the architecture is the kernel. It implements an event queue, over which managers communicate. *Managers* in ENCODERS are responsible for specific tasks such as managing communication interfaces, encapsulating a set of protocols, sending content, and so forth. The managers are represented by the wedges in Figure 1, labeled Forwarding, Node, and so forth. The circular structure of the architecture illustrates its layer-less design, depicting that there is no fixed ordering between the managers.

The managers are responsible for specific tasks and interact only by producing and consuming events. This makes it easy to add and remove managers in the architecture, as they do not directly interact. Managers use *modules* (depicted in the figure as small outermost circles attached to certain managers) to implement specific algorithms (instead of mandating a specific algorithm for each component). Thus the Forwarding Manager has different forwarding algorithms or protocols available in modules.

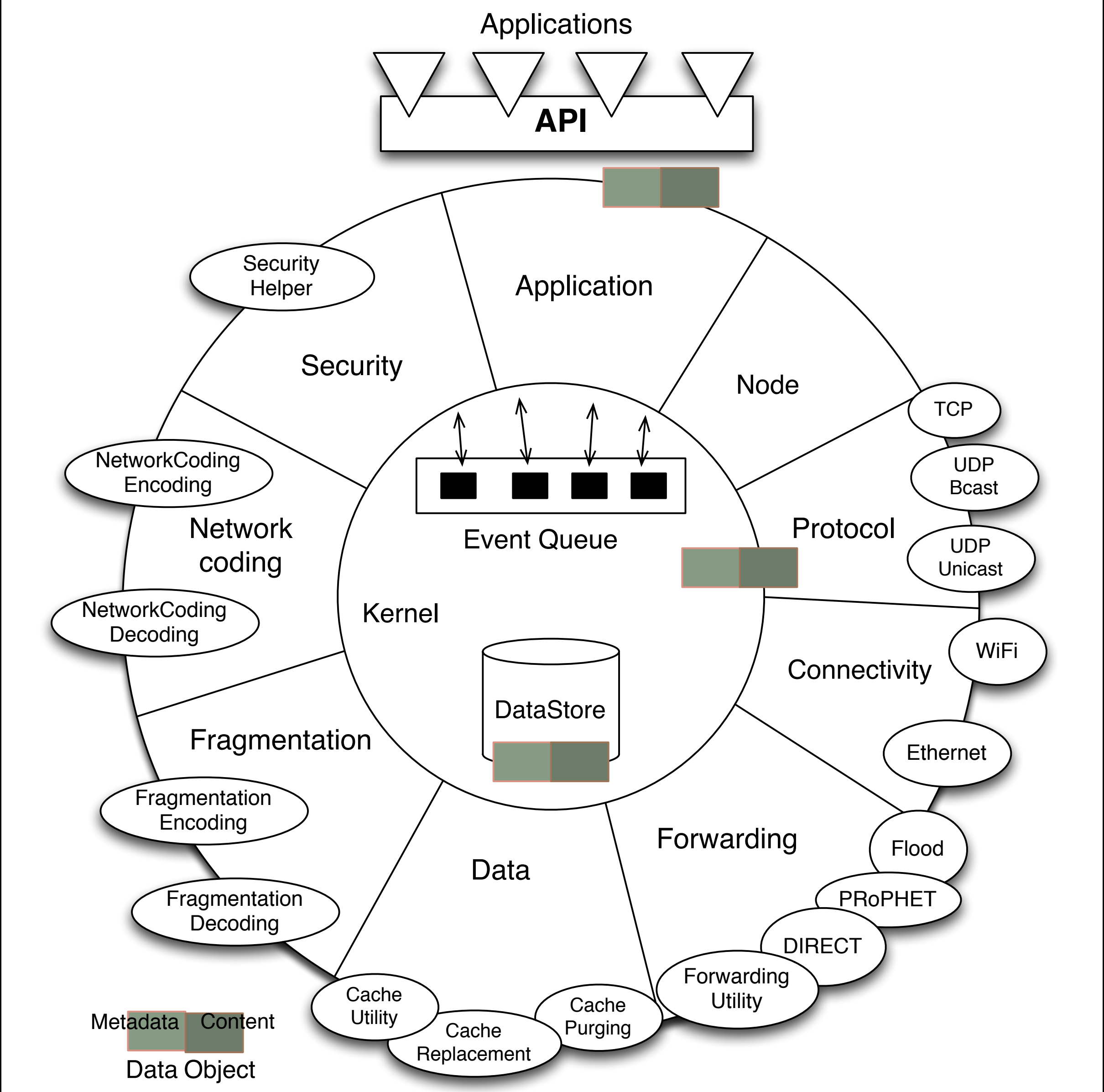

Figure 1. *ENCODERS Architecture and API*. The circle depicts the layer-less interaction. Each wedge of the circle is a *Manager*, and they interact through the Event Queue and DataStore in the Kernel (center). The managers may use the options provided by the modules in the small circles. Not all managers are shown.

2 ENCODERS Functional Areas

The technology development performed by the SRI team can be grouped into the following areas that extend the Haggle framework in orthogonal dimensions. The sections in this document will group configuration options into these areas. The descriptions below introduce concepts that will be used in describing the parameters.

1. **Content and Metadata Distribution.** The Haggle enhancement that provides interest-driven content distribution will be based on DIRECT (Ignacio Solis 2008) that has been developed in the DARPA Disruption-Tolerant Networking (DTN) Program. It extends Haggle by providing a mechanism that supports lightweight dissemination of node descriptions (including node interests) and dissemination of content, which is typically more heavyweight, when new interests are detected. Our new generalized content-distribution architecture supports utility-based dissemination to prioritize the content dissemination for better use of limited network bandwidth and context-awareness.
2. **Content-Based Caching.** The basic cache management strategy is based on opportunistic caching and orderings, ensuring that the purging and replacement strategies are not ad hoc, but inherently content-based, and intuitive concepts such as relative prioritization and utility can be expressed for efficiency and timeliness of content delivery. We think of caching as a specialized utility-optimization algorithm. We have extended Haggle by the notion of configurable caching strategies providing a generalized mechanism for handling data that has that has been marked with specific tags by the user. Utility can also consider military social structures and be used for smart replication to enable cooperative caching in the tactical edge.
3. **Network Coding and Fragmentation.** Network coding is a technique that fragments and encodes content for dissemination in the network. Its purpose is to provide robust (and hopefully efficient) dissemination in networks with unreliable node connectivity. For CBMEN, Haggle is enhanced with a Network-coding Manager that operates on the data content but NOT on the attributes (so that the attributes can still be matched with interests). Network coding is applied selectively, i.e., only to data objects that have content (e.g., with a minimum file size), but not to relatively small node descriptions. ENCODERS also provides the option of fragmentation, which can be used as an alternative to network coding
4. **Security** in ENCODERS provides data integrity, non-repudiation and confidentiality. Our approach replaces the implementation of digital signatures in Haggle with a new multi-authority certification framework for signatures and adds a new layer of attribute-based encryption to cryptographically implement complex security policies framed over attributes certified by multiple authorities.

3 Configuration of Content and Metadata Distribution

Dissemination in ENCODERS involves the Node, Data, Forwarding, and Application Managers. The configuration of each is described in its own subsection. The configuration of content-based protocols, which configure the Protocol and Connectivity Managers, is also important and has been placed in its own section, immediately following this one (Section 4).

SRI modified Haggle to consistently use the mathematical semantics (see the ENCODERS **Software Design Description)** at the application-level, which is what matters for end-users. This feature is called **first-class applications,** and application nodes become first-class citizens (as opposed to be approximately represented by their proxy, the device node). Aggregation is disabled in favor of accuracy.

This section has a subsection for each Manager than can be configured to affect distribution. DIRECT is configured through the <ForwardingManager> and <NodeManager> tags. The Forwarding Manager tag specifies which delegate forwarding modules to use for what content, while the Node Manager tag specifies the node description propagation parameters.

3.1 ForwardingManager

We present an example excerpt from a configuration file for the Forwarding Manager. The XML format that is accepted by the Forwarding Manager can be inferred from this example. This example enables light-weight flooding of node descriptions, proactive replication of small data objects and data objects that are tagged with specific tags declared in the configuration file. This configuration uses DIRECT for specifically tagged data objects and for all other data objects not covered by the previous cases. The value of the “class_name” attribute of the classifiers is the tag that the forwarders can register for through the “contentTag” attribute.

```
<ForwardingManager
    max_nodes_to_find_for_new_dataobjects="30"
    query_on_new_dataobject="true"
    periodic_dataobject_query_interval="0"
    enable_target_generation="false"
    push_node_descriptions_on_contact="true">
  <ForwardingClassifier name="ForwardingClassifierPriority">
   <ForwardingClassifierPriority>
    <ForwardingClassifier name="ForwardingClassifierAttribute" priority="5">
     <ForwardingClassifierAttribute attribute_name="ContentType"
        attribute_value="Direct" class_name="hw" />
    </ForwardingClassifier>
    <ForwardingClassifier name="ForwardingClassifierNodeDescription" priority="4">
     <ForwardingClassifierNodeDescription class_name="lw" />
    </ForwardingClassifier>
    <ForwardingClassifier name="ForwardingClassifierAttribute" priority="3">
     <ForwardingClassifierAttribute attribute_name="ContentType"
        attribute_value="Flood" class_name="lw" />
    </ForwardingClassifier>
    <ForwardingClassifier name="ForwardingClassifierSizeRange" priority="2">
      <ForwardingClassifierSizeRange min_bytes="0" max_bytes="1024"
          class_name="lw" />
    </ForwardingClassifier>
    <ForwardingClassifier name="ForwardingClassifierAllMatch" priority="1">
      <ForwardingClassifierAllMatch class_name="hw" />
    </ForwardingClassifier>
  </ForwardingClassifierPriority>
 </ForwardingClassifier>
 <Forwarder protocol="Flood" contentTag="lw">
  <Flood push_on_contact="true" />
 </Forwarder>
 <Forwarder protocol="AlphaDirect" contentTag="hw" />
</ForwardingManager>
```

Next, we describe each parameter relevant to the Forwarding Manager than an administrator might want to set in the configuration file.

push_node_descriptions_on_contact — true or false, if enabled, this feature allows flooded node descriptions to be propagated to new neighbors that may become connected after the initial flood.

dataobject_retries — The default is 1 to mimic behavior of original Haggle. This applies to data objects that are sent or forwarded to peers (not to applications) that are not node descriptions (see next parameter). Note that these retries are in addition to the retries of the protocol (if any). Note also that node refresh provides another level that can trigger retransmissions (but this does not apply to proactive dissemination).

EXCEPTION: The dataobject_retries parameter does not apply to data objects that use the optimized fast path (see dataobject_retries_shortcircuit below). Proactively disseminated data objects fall into this category.

node_description_retries — The initial transmission (but not the forwarding) of a node description has its own retry count (as part of the Node Manager, see below). This parameter defines the maximum number of retries for forwarding node description (e.g., beyond the first hop). Typically with node refresh, since node descriptions are short-lived (e.g., 30sec) retries are not needed.

max_nodes_to_find_for_new_dataobjects — The default is 30. This is an upper bound for the number of possible (remote) targets that can be generated in response to an incoming data object. If used with the first-class applications feature, it needs to be large enough to include all target applications. Also consider that there might be hidden applications (e.g., in the interest manager) that should be included.

enable_target_generation — The default is true as in Haggle, but we recommend setting this to false. Target generation is a potentially expensive operation that for each incoming data object and each current neighbor generates all targets that can be reached through this neighbor. Without this feature all targets are generated only once for the data object and then the best delegate neighbor is selected based on the result.

dataobject_retries_shortcircuit — Similar to *data_object_retries*, but applied to data objects using the fast path, such as those proactively disseminated by the flood forwarder.

max_forwarding_delay_base, max_forwarding_delay_linear (synonymous to *max_node_desc_forwarding_delay*) - Specifies the maximum forwarding delay in millisec. This number is used to add a randomized delay to data objects (excluding node descriptions) that are sent to multiple nodes at the same time. It is especially important to desynchronize transmissions in clusters and hence to reduce the number of redundant transmissions. The delay will be 0 for objects sent to only one peer and randomly chosen from [0, *max_forwarding_delay_base* + *max_forwarding_delay* * (number of target neighbors - 1)] otherwise. Here, *max_forwarding_delay_base* is a minimum randomized delay that will always be added (independent of the number of target neighbors). The default is 20ms, to randomize the order in the choice of send events and hence in the choice of the (primary) sender protocol in case of broadcast (where subsequent send event are suppressed by Bloom filters). The default for *max_forwarding_delay_linear* is 0. *max_forwarding_delay* is synonymous to *max_forwarding_delay_linear*. Note that additional delays can be specified at the protocol level (see UDP broadcast protocol).

max_node_desc_forwarding_delay_base and *max_node_desc_forwarding_delay_linear* (synonymous to *max_node_desc_forwarding_delay*) are similar parameters for node descriptions. Defaults are 0 and 20ms, respectively.

accept_neighbor_node_descriptions_from_third_party — Using the default setting true node descriptions for neighbors should come from the neighbor directly and are ignored otherwise.

neighbor_forwarding_shortcut — By default this option is true to directly forward data objects to neighbors that are interested, in contrast to let the routing algorithm make the choice (treating it as multi-hop delegation). For best efficiency the recommended setting is true, but there may be reasons to set this option to false in connection with certain security (digital signature) features and the previous option accept_neighbor_node_descriptions_from_third_party.

load_reduction_min_queue_size, load_reduction_max_queue_size — These parameters are used to reduce load (currently probabilistically skipping queries) with the goal to keep kernel event queue size below the max bound (but there is no guarantee). Default is unlimited queue size (i.e., no load reduction).

ForwardingClassifier
This attribute is responsible for specifying a classification module, so that different classes of content can be routed differently. The parameters are as follows:

name - This attribute specifies which classification module to use.

ForwardingClassifierBasic
DEPRECATED — use ForwardingClassifierNodeDescription instead. This classification module classifies content into two categories, light-weight and heavy-weight. Currently, node descriptions are classified as light-weight content, and data objects are classified as heavy weight content. The parameters are as follows:

lightWeightClassName — The "tag" that should be assigned to light-weight content. Forwarder modules are specified on a per-tag basis.

heavyWeightClassName — The "tag" that should be assigned to heavy-weight content. Forwarder modules are specified on a per-tag basis.

ForwardingClassifierNodeDescription
This classification module only tags data objects that contain a node description. The parameters are as follows:

class_name — The "tag" that should be assigned to data objects belonging to a node description.

ForwardingClassifierAttribute
This classification module tags data objects that have a certain attribute (name,value) pair. The parameters are as follows:

attribute_name — The name of the attribute to be tagged.

attribute_value — The value of the attribute to be tagged.

class_name — The "tag" that should be assigned to data objects that have the correct attribute name, value.

ForwardingClassifierSizeRange
This classification module tags data objects that are within a certain size range. The parameters are as follows:

min_bytes — The minimum number of bytes of the data object size to be classified.

max_bytes — The maximum number of bytes of the data object size to be classified.

class_name — The "tag" that should be assigned to the data object that has the correct size.

ForwardingClassifierAllMatch
This classification module tags every data object with a specified tag. It is useful as a catch-all with the priority classifier.

ForwardingClassifierPriority
This classification module is an aggregate of classifiers, organized by their priority, where a higher priority value has more priority. It has no parameters, but expects child tags of the form:

```
<ForwardingClassifier name="ForwardingClassifierX.." priority="...">
   <ForwardingClassifierX... />
</ForwardingClassifier>
```

Where "name" specifies the name of the classifier, and "priority" specifies the priority.

Forwarder
This attribute is responsible for specifying and configuring a forwarder module. A forwarder module is assigned responsibility for a specific tag, where the tag is assigned to the piece of content by the classifier. The parameters are as follows:

protocol — Specifies which forwarder module to load. Currently we support Prophet, AlphaDirect and Flood.

contentTag — Specifies which "tag" (assigned by the classifier) that this specific forwarder module is responsible for.

NOTE: This value can be omitted for one forwarder. In this case, this will become the default forwarder responsible for all content that does not have an existing forwarder responsible for its propagation.

Flood
This forwarder floods a data object to each 1-hop neighbor. Upon insertion of a flooded data object, this module short-circuits the Haggle matching mechanism, and instead attempts to immediately send the data object to each 1-hop neighbor (at the time of receiving the data object). The parameters are as follows:

push_on_contact — "true" or "false": if "true" then all data objects marked for flooding will be sent to a neighbor upon contact, provided that the neighbor does not already have the data object. If set to "false" then a flooded data object will only propagate within the connected component of the publisher, at the time that the data object was published.

enable_delegate_generation — "true" or "false": if "true" then flooding can additionally be triggered by incoming queries from remote nodes. This only makes sense together with the following parameter. Default is "false".

reactive_flooding — "true" or "false": if "true" content is not immediately flooded when injected by an application, but rather waits in the local cache of the publishing node till it is requested, in which case it is flooded, but only if requested by a remote node that is not an immediate neighbor. The default is false (i.e., proactive flooding). If "true", this parameter needs to be used together with enable_delegate_generation = "true".

AlphaDirect
This forwarder uses interest-driven routing inspired by DIRECT to forward data objects along the reverse path of the interest propagation. There are no parameters.

Prophet
This forwarder uses an extension of Haggle's Prophet algorithm to forward data objects according to a topology connectivity matrix.

The variant of Prophet implemented in unmodified Haggle only disseminates routing information when the network changes, as typical for pocket-switched networks. To better deal with typical mobile ad hoc networks and less dynamic topologies, we added a parameter *periodic_routing_update_interval* to the Forwarding Manager to specify the interval for periodic routing updates in seconds (0 means disabled). Another Boolean parameter called sampling enables periodic updates of the Prophet predictabilities by sampling the current neighbor status (rather than updating them only on changes). Finally, a delta parameter has been added as recommended in the Internet draft.

Here is an example configuration that activates Prophet:

```
<ForwardingManager query_on_new_dataobject="true" periodic_dataobject_query_interval="0"
                    recursive_routing_updates="false" periodic_routing_update_interval="10">
        <Forwarder max_generated_delegates="1" max_generated_targets="1" protocol="Prophet">
                <Prophet strategy="GRTR" P_encounter="0.75" alpha="0.5" beta="0.25"
                        gamma="0.999" delta="0.01" aging_time_unit="1" sampling="true" />
        </Forwarder>
</ForwardingManager>
```

Example forwarder policies for interest data objects are as follows:

```
<ForwardingManager
  max_nodes_to_find_for_new_dataobjects="30"
  max_forwarding_delay="2000"
  node_description_retries="0"
  dataobject_retries="1"
  dataobject_retries_shortcircuit="2"
  query_on_new_dataobject="true"
  periodic_dataobject_query_interval="0"
  enable_target_generation="false"
  push_node_descriptions_on_contact="false"
  load_reduction_min_queue_size="500"
  load_reduction_max_queue_size="1000"
  enable_multihop_bloomfilters="false">
 <ForwardingClassifier
    name="ForwardingClassifierAttribute">
  <ForwardingClassifierAttribute
     class_name="flood"
     attribute_name="Interests"
```

```
      attribute_value="Aggregate" />
  </ForwardingClassifier>
  <Forwarder
     protocol="Flood"
     contentTag="flood" />
  <Forwarder
     protocol="AlphaDirect" />
</ForwardingManager>
```

enable_multihop_bloomfilters — "true" or "false": This is a new parameter in Version 2 that should *always* be set to false when using the InterestManager, as the ForwardingManager is no longer responsible for propagating node descriptions.

This configuration will propagate interest data objects epidemically.

3.2 NodeManager

Below is an excerpt from config.xml that enables node refresh:

```
<NodeManager>
        <Node matching_threshold="0" max_dataobjects_in_match="10"/>
        <NodeDescriptionRetry retries="3" retry_wait="10.0"/>
        <NodeDescriptionRefresh refresh_period_ms="30000" refresh_jitter_ms="1000" />
        <NodeDescriptionPurge purge_max_age_ms="90000" purge_poll_period_ms="30000" />
</NodeManager>
```

The XML format that is accepted by the Node Manager can be inferred from this and later examples. Next, we describe each parameter relevant to the Node Manager than an administrator might want to set in the configuration file.

NodeDescriptionRefresh
This attribute is responsible for specifying how frequently the node descriptions (and the corresponding interests) are propagated through the network. Omitting this tag disables this refresh mechanism. The parameters are as follows:

refresh_period_ms — Specifies how frequently to send a new node description, in milliseconds.

refresh_jitter_ms — A number is picked uniformly at random from the interval [0, refresh_jitter_ms) and added to the period. The purpose of adding this value is to prevent synchronized floods of node descriptions.

NodeDescriptionPurge
This attribute is responsible for specifying when to purge stale node descriptions from the cache, which prevents DIRECT from forwarding on invalid paths (paths that have not recently been refreshed). Omitting this tag disables this purging mechanism.

purge_max_age_ms — Specifies the age for a node description after which it is eligible for purging (in milliseconds). No node description that is younger than this age will be purged by this mechanism.

purge_poll_period_ms — Specifies how frequently node descriptions should be checked for expiration, in milliseconds. A higher frequency means that nodes eligible for purging will be purged sooner, but at the expense of higher CPU due to more events. Conversely, a lower frequency means that nodes will be checked for purging less often, but with lower CPU utilization.

Due to the increased database utilization from increased propagation of node descriptions, we have added an optimization, so called "in memory node descriptions", that does not put node descriptions in the database. This is an option for the Data Manager, and is described in the next section.

In unmodified Haggle, each node description has a *NodeDescription*=<nodeid> attribute (with weight 1), which is used by Haggle internally to detect node descriptions. By abstracting from the specific <nodeid>, using the setting *node_description_attribute*="type", the node description attribute can also be used to ensure that node descriptions of peers overlap (like common interest), which enables them to propagate through the network with the unmodified Haggle routing mechanism. Here is such a configuration:

```
<NodeManager>
        <Node matching_threshold="0" max_dataobjects_in_match="10"
                node_description_attribute="type"/>
        <NodeDescriptionRetry retries="3" retry_wait="10.0"/>
</NodeManager>
```

In this context, it is also possible to use node_description_weight="0" to prevent node descriptions to interfere with the application semantics. This is useful together with the first-class application feature.

ENCODERS uses the NodeDescription attribute when node descriptions are stored in the database like other objects (as in Haggle) but this attribute is not needed when they are kept in memory (a new capability in ENCODERS that is configured with in_memory_node_descriptions). We recommend eliminating this attribute to save bandwidth using node_description_attribute="none" as below:

```
<NodeManager>
        <Node matching_threshold="0" max_dataobjects_in_match="10"
                node_description_attribute="none"/>
        <NodeDescriptionRetry retries="3" retry_wait="10.0"/>
</NodeManager>
```

The following NodeManager settings are recommended for 1-Hop Bloom Filters:

```
<NodeManager>
  <Node
     matching_threshold="0"
     max_dataobjects_in_match="10"
     node_description_attribute="none"
     node_description_attribute_weight="0"
     send_app_node_descriptions="false" />
  <NodeDescriptionRetry
     retries="0"
     retry_wait="10.0"/>
</NodeManager>
```

send_app_node_descriptions — true or false. This is a new parameter in Version 2 that disables forwarding of application-node descriptions by the NodeManager.

These settings will disable node description purging (the Connectivity timeouts will be used to remove stale 1-hop neighbors), and not propagate application-node descriptions. This functionality is the responsibility of the InterestManager.

3.3 DataManager

Haggle's SQLite database is a performance bottleneck and should be disabled if not needed. To this end, a Boolean parameter *use_in_memory_database* has been added to the SQLDataStore. Even if disabled, a data base file is read at startup and written at shutdown, but not used during normal operation. The speedup is significant, and hence this setting is strongly recommended. The SQLite journaling mode can be specified by another parameter journal_mode (which can be off, memory, persist, or truncate). See SQLite documentation for details.

Below is a typical configuration excerpt. It enables the in-memory database without journaling. The XML format that is accepted by the Data Manager can be inferred from this example.

```
<DataManager set_createtime_on_bloomfilter_update="true">
        <Aging period="3600" max_age="86400"/>
        <Bloomfilter default_error_rate="0.01" default_capacity="2000"/>
        <DataStore>
                <SQLDataStore use_in_memory_database="true" journal_mode="off" />
        </DataStore>
</DataManager>
```

Node descriptions are not of importance to applications and hence should not be counted when imposing a bound on the number of data objects returned by a query/subscription. To this end, it is possible to set *count_node_descriptions*="false" as in the excerpt below:

```
<DataManager set_createtime_on_bloomfilter_update="true">
        <Aging period="3600" max_age="86400"/>
        <Bloomfilter default_error_rate="0.01" default_capacity="2000"/>
        <DataStore>
                <SQLDataStore use_in_memory_database="true" journal_mode="off"
                count_node_descriptions="false"/>
        </DataStore>
 </DataManager>
```

Due to the increased database utilization from increased propagation of node descriptions, we have added an optimization, "in-memory node descriptions", which disables insertion of node descriptions in the database. Below is an excerpt from config.xml that demonstrates how to enable this optimization. By default, when not specified, this parameter is false. We recommend always enabling this optimization when appropriate (see below), except in small networks for testing purposes.

```
<DataManager set_createtime_on_bloomfilter_update="true"
                periodic_bloomfilter_update_interval="60">
        <Aging period="3600" max_age="86400"/>
        <Bloomfilter default_error_rate="0.01" default_capacity="2000"/>
        <DataStore>
                <SQLDataStore use_in_memory_database="true" journal_mode="off"
                                in_memory_node_descriptions="true" />
        </DataStore>
</DataManager>
```

in_memory_node_descriptions — When set to "true", this optimization will not insert node descriptions in the database. Note that this optimization should only be used in conjunction with the forwarder flood mechanism for light weight content, because the database resolution operation is short-circuited. In other words, this disables matching node descriptions as data objects to targets.

periodic_bloomfilter_update_interval — Allows the user to specify, in seconds, how often to take non-counting abstractions of the counting Bloom filter, and update it. This replaces potentially invalid (stale) state in the non-counting Bloom filter.

The node store maintains two Bloom filter abstractions per peer to maintain a certain degree of continuity when the abstractions are updated. This feature is turned on by default (continuousBloomfilters is true). In this case, the local Bloomfilter abstraction is replaced only if a minimum time has passed (continuousBloomfilterUpdateInterval with a default of 5000ms) and is updated by a merge (i.e., set union) otherwise.

Database timeout for CacheReplacementTotalOrder is configurable. The default values are shown below.

```
<DataManager>
      <DatabaseTimeout database_timeout_ms="1500"
                drop_new_object_when_database_timeout="true"/>
  </DataManager>
```

Example caching policies for 1-Hop Bloom Filters are as follows:

```
<CacheStrategy name="CacheStrategyUtility">
 <CacheStrategyUtility
   knapsack_optimizer="CacheKnapsackOptimizerGreedy"
   global_optimizer="CacheGlobalOptimizerFixedWeights"
   utility_function="CacheUtilityAggregateMin"
   max_capacity_kb="10240"
   watermark_capacity_kb="5120"
```

```
    compute_period_ms="200"
    purge_poll_period_ms="1000"
    purge_on_insert="true"
    publish_stats_dataobject="false"
    keep_in_bloomfilter="true"
    handle_zero_size="true">
   <CacheKnapsackOptimizerGreedy />
   <CacheGlobalOptimizerFixedWeights
     min_utility_threshold="0.1">
    <Factor
      name="CacheUtilityAggregateMin"
      weight="1" />
    <Factor
      name="CacheUtilityReplacementTotalOrder"
      weight="1" />
    <Factor
      name="CacheUtilityPurgerAbsTTL"
     weight="1" />
   </CacheGlobalOptimizerFixedWeights>
   <CacheUtilityAggregateMin
     name="CacheUtilityAggregateMin">
    <Factor
      name="CacheUtilityReplacementTotalOrder">
     <CacheUtilityReplacementTotalOrder
       metric_field="SeqNo"
       id_field="Origin"
       tag_field="Interests"
       tag_field_value="Aggregate" />
    </Factor>
    <Factor
      name="CacheUtilityPurgerAbsTTL">
     <CacheUtilityPurgerAbsTTL
       purge_type="ATTL_MS"
       tag_field="Interests"
       tag_field_value="Aggregate" />
    </Factor>
   </CacheUtilityAggregateMin>
  </CacheStrategyUtility>
</CacheStrategy>
```

This will use total-order replacement on interest data objects (fresher interest data objects replace staler interest data objects), and it will use the absolute TTL specified in the InterestManager settings to purge old interest data objects.

3.4 ApplicationManager

Each node can be equipped with default interest, e.g., to enable proactive propagation of certain data objects (including node descriptions).

NOTE: This function is not needed if such data objects are already proactively disseminated using the Flood forwarder (the recommended configuration).

The default interests are added to the device node description, but not to the application node description running on this device. Hence, even with default interests, applications have to perform an explicit subscription to fetch the data from the local cache.

Typical configuration excepts are:

```
<ApplicationManager>
        <Attr name="RegistrarMetadata">description</Attr>
</ApplicationManager>
```

```
<ApplicationManager>
         <Attr name="CommonInterest" weight="100">true</Attr>
</ApplicationManager>
```

The first example may be useful to proactively disseminate rich metadata objects. The second example just establishes common interest so that device node descriptions have overlapping attributes and are proactively exchanged. Multiple attributes can be similarly specified.

We strongly suggest using the following configuration for 1-Hop Bloom Filters, which will enable triggering interest refresh when applications deregister and subsequently register.

```
<ApplicationManager delete_state_on_deregister="true" />
```

3.5 SendPriorityManager

An example Send Priority configuration is:

```
<SendPriorityManager
 enable="true"
 run_self_tests="false"
 partial_order_class="PartialOrderCombinerOrdered">
  <PartialOrderCombinerOrdered
   high_class_name="PartialOrderAttribute"
   high_class_param_name="PartialOrderAttribute1"
   low_class_name="PartialOrderCombinerOrdered"
   low_class_param_name="PartialOrderCombinerOrdered1">
    <PartialOrderAttribute1
     attribute_name="bolo" />
    <PartialOrderCombinerOrdered1
     high_class_name="PartialOrderPriorityAttribute"
     high_class_param_name="PartialOrderPriorityAttribute1"
     low_class_name="PartialOrderFIFO"
     low_class_param_name="PartialOrderFIFO1" >
      <PartialOrderPriorityAttribute1
       priority_attribute_name="priority" />
      <PartialOrderFIFO1 />
    </PartialOrderCombinerOrdered1>
  </PartialOrderCombinerOrdered>
</SendPriorityManager>
```

A total order is defined by combining multiple partial orders. The order is as follows:

. PartialOrderAttribute with attribute 'bolo'
. PartialOrderPriorityAttribute with attribute 'priority'
. PartialOrderFIFO

For example, consider the following 5 data objects that are queued in the following order to be sent to a neighbor:

a. D1: { "bolo" : "", "priority" : 1 }
b. D2: { "bolo" : "", "priority" : 2 }
c. D3: { "bolo" : "" }
d. D4: { "priority" : 1 }
e. D5: { "priority" : 2 }

Based on this partial order, they will be ordered to send as follows:

D2, D1, D3, D5, D4

Note that D3 is prioritized before D5 and D4 due to the PartialOrderFIFO attribute.

The following `SendPriorityManager` configuration options are supported:

enable — Set to `true` to enable the `SendPriorityManager`, `false` otherwise.

debug — Set to `true` to enable verbose debug messages, `false` otherwise.

parallel_factor — The number of concurrent data objects to be queued for a particular receiver.

run_self_tests — Set to `true` to run unit tests on manager initialization, `false` otherwise.

partial_order_class — The class name of the partial order class to load. The configuration options for the class are loaded by looking for a tag with the same name within the `SendPriorityManager` tag.

The following `PartialOrder` classes are supported and their options are supported:

. **PartialOrderAttribute** — data objects with the specified attribute have priority (>) over data objects without the attribute. Data objects either both with the attribute, or without the attribute are incomparable (|).

attribute_name — specifies the attribute name that gives priority to data objects with this given attribute (over data objects without this attribute).

. *PartialOrderPriorityAttribute* — uses the integer value of attributes with a specific name to order said data objects--higher values have greater priority than lower values. If one (or both) of the data objects do not have the specified attribute, then they are incomparable (|). One could configure this partial order to respect the node's role: for example, a UAV may have a configuration setting that gives high priority to squad leader content to improve squad-leader communication, whereas within a squad blue force tracking might have higher priority for squad members.

priority_attribute_name — Set to the attribute name whose value is interpreted as an integer for prioritization.

. *PartialOrderFIFO* — totally orders the data objects by the queue insertion date, where data objects that were inserted earlier have priority over data objects that are inserted later.

. *PartialOrderCombinerOrdered* — Combines two PartialOrder's to construct a new PartialOrder, where one partial order is consulted before the second partial order when ordering two data objects. When two data objects are compared, if the first partial order (high priority partial order) has a result other than |, then that result is returned. Otherwise, the result of the second partial order (low priority partial order) is returned.

high_class_name — The high priority partial order class name.

high_class_param_name — The name of the sub-tag which contains the parameters to initialize the high priority partial order.

low_class_name — The low priority partial order class name.

low_class_param_name — The name of the sub-tag which contains the parameters to initialize the low priority partial order.

3.6 ReplicationManager

An example Utility-based Replication configuration is:

```
<ReplicationManager name="ReplicationManagerUtility" >
  <ReplicationManagerUtility
    enabled="true"
    run_self_test="false"
    forward_poll_period_ms="40000"
    knapsack_optimizer="ReplicationKnapsackOptimizerGreedy"
    global_optimizer="ReplicationGlobalOptimizerFixedWeights"
    utility_function="ReplicationUtilityAggregateMin" >
  <ReplicationGlobalOptimizerFixedWeights min_utility_threshold="0.1" >
    <Factor name="ReplicationUtilityAggregateMin" weight="1" />
    <Factor name="ReplicationUtilityWait" weight="1" />
  </ReplicationGlobalOptimizerFixedWeights>
  <ReplicationKnapsackOptimizerGreedy discrete_size="1"  />
  <ReplicationUtilityAggregateMin name="ReplicationUtilityAggregateMin">
```

```
        <Factor name="ReplicationUtilityWait" >
          <ReplicationUtilityWait name="ReplicationUtilityWait"/>
        </Factor>
    </ReplicationUtilityAggregateMin>
    </ReplicationManagerUtility>
</ReplicationManager>
```

The following `ReplicationManagerUtility` configuration options are supported:

enable — Set to `true` to enable the `ReplicationManagerUtility`, `false` otherwise.

run_self_test — Set to true to enable internal verification. This only tests individual components, not the full functionality. The results are output into the log.

replication_cooldown_ms —
This option sets how long to wait, before the knapsack will be used to replicate again (in milliseconds).

max_repl_tokens — This sets how many tokens, maximum. This affects either how many data objects, total, or how many data objects, per node, to be considered for the knapsack.

token_per_node — This sets either the tokens to be counted per node (true), or, as data objects total (false).

forward_poll_period_ms — This sets the timer (in milliseconds) for how often to execute the replication utility code. The replication code will always be called on a new data object (data object inserted) or a change in neighbor nodes (a new node seen, a node is no longer in 1-hop neighborhood, or a node change, e.g., updated contact times).

knapsack_optimizer — Specify which knapsack optimizer, which will decide, based upon file size, utility value, and estimated amount of time to replicate, which data objects to prioritize. Currently, there is only one knapsack, `ReplicationKnapsackOptimizerGreedy`, is supported.

global_optimizer — This option sets the global optimizer. Currently, only `ReplicationGlobalOptimizerFixedWeights` is supported.

utility_function — Specifies the title of xml code which will be used in utility replication, and which ReplicationUtilityAggregate[Minimum, Sum, Maximum] will be used.

The following 'ReplicationGlobalOptimizerFixedWeights` configuration options are supported, which function similar to CacheGlobalOptimizerFixedWeights.
The difference is, CacheGlobalOptimizerFixedWeights purges data objects if its utility is below a certain threshold, while ReplicationGlobalOptimizerFixedWeights considers replicating data objects if their utility is above a certain threshold. All data objects above a threshold are passed to the knapsack, to optimize which data objects are sent first.

min_utility_threshold —The minimum utility value considered for replication.

Within `ReplicationGlobalOptimizerFixedWeights`:
<Factor name=[utility] weight="x"/>

This gives each 'utility' function a weight of 'x'. Valid utilities are:

f. ReplicationUtilityAggregateMin
g. ReplicationUtilityAggregateMax
h. ReplicationUtilityAggregateSum
i. ReplicationUtilityNOP
j. ReplicationUtilityRandom
k. ReplicationUtilityAttribute
l. ReplicationUtilityWait
m. ReplicationLocal
n. ReplicationNeighborhoodOtherSocial

ReplicationUtilityWait : This utility function gives a value of 0, until a specified amount of time has passed, in which case, it returns 1.

The following options are supported for `ReplicationUtilityWait`:

wait_s — Specifies, in seconds, how long this function should return zero. After this time interval has expired, return a value of '1'.

Refer to READEME-ucaching.txt for other utility functions.

The following ReplicationKnapsackOptimizerGreedy configuration options are supported:

discrete_size — Specifies a block size to be used for consideration of the knapsack.

3.7 InterestManager

An example Interest Manager configuration is:

```
<InterestManager
  enable="true"
  interest_refresh_period_ms="30000"
  interest_refresh_jitter_ms="500"
  absolute_ttl_ms="90000" />
```

Here, interests will be periodically refreshed after 30 seconds, with half a second of jitter. After 90 seconds the interests will be removed (if the appropriate caching settings are enabled).

The following InterestManager configuration options are supported:

enable — Set to `true` to enable the interest manager *interest_refresh_period_ms* — Interest refresh period in milliseconds.

interest_refresh_jitter_ms — Interest jitter in milliseconds.

absolute_ttl_ms — Absolute ttl in milliseconds for an interest data object.

debug —
Set to `true` to enable extra verbose debug messages.

consume_interest — Set to `true` to enable removing an interest upon matching and forwarding. This feature only applies to interests that have a maximum match of 1 (`-m 1` option to `haggletest`).

run_self_tests — Set to `true` to run basic unit tests.

4 Configuration of Content-based Protocols

To improve performance, ENCODERS refactored the Haggle protocols to implement additional UDP-based protocol options. The configuration of content-based protocols, which configure the Protocol and Connectivity Managers, is described in this section.

The stop-and-wait protocol is described in the Software Design Document. We have parameterized many of the timers and retry counts used by this protocol, enabling users to specify them in the configuration file.

Below is an example excerpt from the configuration file that enables broadcast of node descriptions, UDP unicast for small data objects, and TCP for all other data objects. Also, it uses the "arphelper" program for manual arp insertion. Note that the *maxSendTimeouts* option specifies the maximum number of attempts the protocol will have to send the data object.

```
<ProtocolManager>
  <ProtocolClassifier name="ProtocolClassifierPriority">
    <ProtocolClassifierPriority>
      <ProtocolClassifier name="ProtocolClassifierNodeDescription" priority="3">
        <ProtocolClassifierNodeDescription outputTag="nd" />
      </ProtocolClassifier>
      <ProtocolClassifier name="ProtocolClassifierSizeRange" priority="2">
        <ProtocolClassifierSizeRange minBytes="0" maxBytes="4416" outputTag="lw" />
      </ProtocolClassifier>
      <ProtocolClassifier name="ProtocolClassifierAllMatch" priority="1">
        <ProtocolClassifierAllMatch outputTag="hw" />
      </ProtocolClassifier>
    </ProtocolClassifierPriority>
  </ProtocolClassifier>
  <Protocol name="ProtocolUDPBroadcast" inputTag="nd">
    <ProtocolUDPBroadcast
      use_arp_manual_insertion="true" arp_manual_insertion_path="/etc/arphelper" />
    </Protocol>
    <Protocol name="ProtocolUDPUnicast" inputTag="lw">
      <ProtocolUDPUnicast maxSendTimeouts="10" />
    </Protocol>
    <Protocol name="ProtocolTCP" inputTag="hw">
      <ProtocolTCP backlog="30" />
    </Protocol>
</ProtocolManager>
<ConnectivityManager use_arp_manual_insertion="true"
  arp_manual_insertion_path="/etc/arphelper" />
```

4.1 Parameters for Protocols

We describe each parameter relevant to protocols than an administrator might want to set in the configuration file. First, we describe generic parameters that apply to all protocols.

waitTimeBeforeDoneMillis — The maximum number of milliseconds that a protocol sender or receiver will wait for either data to send or data to receive. A high value is useful to avoid protocol instance churn and the expense of possibly maintaining stale protocols, while a low value by prematurely remove fresh protocols but keep lower state. The default is 60000ms.

passiveWaitTimeBeforeDoneMillis — Similar to previous parameter, but for passive UDP receivers, which should be quickly terminated if not needed. The default is 10000ms. Only relevant for UDP broadcast with control (see below).

connectionAttempts — The maximum number of times to try connecting to a peer (used mainly by TCP). Default is 4.

maxBlockingTries — The maximum number of times to try receiving on the receive socket, and having it block. Default is 5.

blockingSleepMillis — The number of seconds to wait when the receive socket is blocked, waiting to receive data. Default is 400ms.

connectionWaitMillis — The maximum number of milliseconds to wait when establishing a connection (used mainly by TCP). Default is 20000ms.

connectionPauseMillis — The number of milliseconds to pause before trying to connect again, upon a failure. Default is 5000ms.

connectionPauseJitterMillis — The number of milliseconds to add to the connectionPauseMillis, drawing uniformly from [0, connectionPauseJitterMillis). Default is 20000ms.

maxProtocolErrors — The maximum number of protocol errors allowable before closing and deleting the protocol. Default is 4.

maxSendTimeouts — The maximum number of times to retry sending a data object, upon send failure. The default is 0. Not recommended for use with TCP.

load_reduction_min_queue_size and *load_reduction_max_queue_size* — These per-protocol parameters enable a simple probabilistic load reduction mechanism which probabilistically discards data objects to keep queue size below maximum (but without guarantee). The default queue size is unlimited (i.e. no load reduction).

Parameters for TCP
port — The TCP port used by the protocol. The default is 9697.

backlog — The number of entries that can fit in the listen queue (effectively the maximum number of concurrent TCP sessions on this port).

It is possible to configure multiple instances of the same protocol type, but the port numbers must not conflict. A second instance of TCP can be specified as follows:

```
<Protocol name="ProtocolTCP" inputTag="tcp2">
        <ProtocolTCP port="9698" />
</Protocol>
```

Parameters for UDP Broadcast
arp_manual_insertion — true or false, manually insert an ARP entry if it is missing and the protocol receives a node description with the MAC address.

arp_manual_insertion_path — system path, location of the compiled arphelper program which wraps "arp -s" and has setuid root.

useArpHack — true or false, issue a ping in the event that the ARP cache entry is missing for the destination, in order to indirectly trigger an ARP request.

Parameters for UDP Broadcast and Unicast
use_control_protocol — "true" or "false" to enable/disable the control protocol.

control_port_a and *control_port_b* — The ports that the sender and receiver uses to exchange control messages (in connection with the control protocol).

no_control_port — The port that is used to exchange data messages representing metadata and payload of data objects. The defaults for these ports are 8791-8793 for UDP broadcast and 8788-8790 for UDP unicast, respectively.

For both, TCP and UDP, it is possible to configure multiple instances of the same protocol type, but the port numbers must not conflict, hence they need to be explicitly specified for each additional instance. For instance, for a second instance of UDP broadcast you can use:

```
<Protocol name="ProtocolUDPBroadcast" inputTag="bcast2">
<ProtocolUDPBroadcast
        control_port_a="8794" control_port_b="8795" no_control_port="8796" />
</Protocol>
```

Limiting Protocol Instances
Too many protocol instances can lead to resource bottlenecks and potential fatal errors (e.g., due to lack of memory or file descriptors). Hence, the number of instances should be conservatively limited. For TCP protocols it is advisable to use a small number of sender instances per link together with suitable kernel setting to allow quick termination of protocols in case of disruptions. For example, the kernel setting

```
        sysctl -w net.ipv4.tcp_syn_retries=0
```

reduces the timeout for unsuccessful TCP connections to ~10 sec instead of the usual 180s with 5 retries.

maxInstances — Maximum number of sender protocol instances on a single node (default 100). Relevant for TCP and UDP protocols.

maxInstancesPerLink — Maximum number of sender protocol instances for each link, i.e., pair of nodes (default 3). Relevant for TCP and UDP protocols. For UDP-based protocols there is no significant startup/shutdown time, hence we recommend to limit the instances to one as exemplified in the configuration except below.

maxReceiverInstances — Maximum number of receiver instances for UDP protocols (default 100). Relevant for UDP protocols.

maxReceiverInstancesPerLink — Maximum number of sender protocol instances for each link, i.e. pair of nodes (default 1). Relevant for UDP protocols.

maxPassiveReceiverInstances — Maximum number of passive receiver instances for UDP broadcast protocols with control (default 100).

maxPassiveReceiverInstancesPerLink — Maximum number of receiver protocol instances for each link (between neighbors), for UDP broadcast protocols with control (default 40). It is not recommended to increase this in the current version.

NOTE: Currently, no attempt is made to limit the number of neighbors in dense networks. If the system conducts an EMERGENCY shutdown, a possible cause is that the number of active protocol instances is too large (e.g., due to a large number of neighbors). In this case lower limits should be used.

Limiting Protocol Sending Rates
UDP protocols without the control protocol do not have explicit flow control. Hence, it is advisable to configure a maximum rate for data object transmission to reduce the likelihood of packet losses due to buffer overflows. Even in case of TCP and UDP with control, the sending rate for certain kinds of traffic (e.g., node descriptions) can be further limited using the following parameters.

minSendDelayBaseMillis — The minimum time in ms between consecutive data objects transmitted by the same protocol instance.

minSendDelayLinearMillis, *minSendDelaySquareMillis* — Additional delays can be specified by these coefficients to be linear or quadratic in the number of neighbors.

To use these parameters it is important to understand that for UDP broadcast each neighbor gives rise to an independent protocol instance, hence specifying 10000ms for minSendDelayBaseMillis leads to an average delay of 1000ms if there are 10 neighbors and the 10 corresponding protocol instances are used concurrently. In case of broadcast, each transmission may be overheard by many nodes. The following sample excerpt shows how to configure a neighbor-dependent rate limit for UDP broadcast.

```
<Protocol name="ProtocolUDPBroadcast" inputTag="bcast">            <ProtocolUDPBroadcast
          waitTimeBeforeDoneMillis="60000"
          use_arp_manual_insertion="true"
          maxInstancesPerLink="1"
          minSendDelayBaseMillis="1000"
          minSendDelayLinearMillis="100"
          minSendDelaySquareMillis="10" />
</Protocol>
```

maxRandomSendDelayMillis — Random delay in ms just before sending a data object (default is 100ms). This reduces the likelihood of redundant UDP broadcast transmissions (which are suppressed by local peer Bloom filter abstractions). See also maxForwardingDelay for a similar delay in the Forwarding Manager.

4.2 Connectivity Manager Parameters

The Connectivity Manager broadcasts periodic beacons using UDP broadcast (on a reserved port that is not used for the transport protocols) and maintains the current neighborhood. The default parameters are as follows:

```
<ConnectivityManager>
          <Ethernet beacon_period_ms="5000" beacon_jitter_ms="1000"
                    beacon_epsilon_ms="1000" beacon_loss_max="3" />
</ConnectivityManager>
```

The parameters *beacon_period_ms* and *beacon_jitter_ms* define the beaconing frequency and optional jitter added for each transmission. *beacon_loss_max* is the number of tolerated losses and *beacon_epsilon_ms* is an additional error allowed for delayed beacons.

In a high-loss environment (which could be caused by contention or fast-paced mobility), it is advisable to increase the beaconing frequency and the number of tolerated losses. For example:

```
<ConnectivityManager>
          <Ethernet beacon_period_ms="2500" beacon_jitter_ms="500"
                    beacon_epsilon_ms="500" beacon_loss_max="6" />
</ConnectivityManager>
```

Additional Connectivity Manager parameters for use with UDP Protocols are:
use_arp_manual_insertion — true or false, enables or disables ARP insertion when discovering neighbors. Useful in conjunction with ProtocolUDPUnicast and ProtocolUDPBroadcast.

arp_manual_insertion_path — system path, location of the compiled arphelper program which wraps "arp -s" and has setuid root. Used if use_arp_manual_insertion is true.

5 Configuration of Caching

To solve the limitations of unmodified Haggle, SRI added a generic module to the Data Manager, the Cache Strategy Module, which currently has cache replacement and cache purger sub-components. The Cache Strategy Module is responsible for handling data that has been marked with specific tags by the user, through use of attributes.

Administrators will use the configuration file to specify the Cache Strategy Module, and its respective components. The cache replacement and purger submodules specify the tag that identifies the class of content that should be handled by the module. Upon receiving a new data object, the Data Manager will determine if the cache strategy is responsible for the data object, which in turn will query the cache replacement and the cache purger.

The caching strategy is configured by the <CacheStrategy> tag in the configuration for the Data Manager (see Section 3.3). An example is shown below. Note that (i) under the <CachePurgerParallel> tag, two purgers are configured in parallel based on absolute and relative expiration times, and (ii) under the <CacheReplacementPriority> tag, the lexicographical ordering is configured using the priority fields in <CacheReplacement> tags.

```
<DataManager set_createtime_on_bloomfilter_update="true" ">
  <Bloomfilter default_error_rate="0.01" default_capacity="2000"/>
  <DataStore> … </DataStore>
 <CacheStrategy name="CacheStrategyReplacementPurger">
   <CacheStrategyReplacementPurger purger="CachePurgerParallel"
          replacement="CacheReplacementPriority">
      <CachePurgerParallel>
         <CachePurger name="CachePurgerAbsTTL">
            <CachePurgerAbsTTL purge_type="purge_by_timestamp"
                   tag_field="ContentType" tag_field_value="DelByAbsTTL"
                   keep_in_bloomfilter="false" min_db_time_seconds="4.5" />
         </CachePurger>
         <CachePurger name="CachePurgerRelTTL">
             <CachePurgerRelTTL purge_type="purge_after_seconds"
                   tag_field="ContentType" tag_field_value="DelByRelTTL"
                   keep_in_bloomfilter="false" min_db_time_seconds="3.5" />
          </CachePurger>
      </CachePurgerParallel>
   <CacheReplacementPriority>
      <CacheReplacement name="CacheReplacementTotalOrder" priority="2">
         <CacheReplacementTotalOrder metric_field="MissionTimestamp"
                   id_field="ContentOrigin" tag_field="ContentType" tag_field_value="TotalOrder" />
      </CacheReplacement>
      <CacheReplacement name="CacheReplacementTotalOrder" priority="1">
            <CacheReplacementTotalOrder metric_field="ContentCreateTime"
                   id_field="ContentOrigin" tag_field="ContentType" tag_field_value="TotalOrder" />
         </CacheReplacement>
      </CacheReplacementPriority>
   </CacheStrategyReplacementPurger>
</CacheStrategy>
</DataManager
```

Below is a meta-configuration that demonstrates almost all of the utility-based caching features. We then discuss the utility function that it defines. This is mainly for illustrative purposes, and would not necessarily be an effective utility function (for example, a function with a weight of 0 (or a NOP function) simply returns 0 (returns 1), making it redundant, and a min function within a min is redundant).

This defines a utility function as follows:

```
utility(d) =
 min(
    max(0.3*nbrhood(d) + 0.10*random(d) + 0.70*pop(d) + 0*NOP, newtimeimmune(d)),
    min(relttl(d), absttl(d)),
    replacement(d)    )
```

Each field indicates as follows:

nbrhood	- corresponds to the defined neighborhood function
random	- corresponds to the defined random function
pop	- corresponds to the defined LRU_K/LRFU function
NOP	- corresponds to the NOP function (returns 1)
relttl	- corresponds to relative TTL purging
absttl	- corresponds to absolute TTL purging
replacement	- corresponds to total order replacement

Recall that every utility function returns a value between 0 or 1, and utility(d) is simply a utility function composed of other utility functions. 0 indicates that the data object has no value and should be evicted from the cache, while 1 means that the data object has the highest possible value.

The threshold and constants set by the global optimizer allow utility caching to fine tune the influence of each utility function. Here, any data object d which has utility(d) < 0.1 will be evicted immediately. Note that a weight of 0 simply disables the utility function (here the NOP function is disabled). By using the max/min functions, one can create "overrides" that allow a utility function to ignore the utilities of other utility functions. In this configuration, if either replacement or the purgers (relative ttl, rel ttl, or absolute ttl, abs ttl) return a 0, then the entire utility function will return 0 (since they are wrapped in a min() function). If none of them return 0 (they only return 0 if they believe the DO should be evicted, otherwise they return 1), then the max function is used.

The max function in this example allows the TimeImmunity utility function (newtimeimmune) to override the utility from the summation function. This is useful to allow a data object that is not purged by replacement or time purging to remain in the cache for at least a specified amount of time (12 seconds in this case). Once the time immunity has expired, then the summation utility function is executed. The summation function here takes the values of neighborhood (nbrhood), random, LRFU (pop) and

NOP, and sums them to construct a single utility value. The weights specified by the global optimizer limits the amount of "influence" one of these functions have on the entire value. For example, the global optimizer gives NOP a value of 0 to disable it (it simply returns 1), while popularity has the most influence.

```
<CacheStrategy name="CacheStrategyUtility">
  <CacheStrategyUtility knapsack_optimizer="CacheKnapsackOptimizerGreedy"
        global_optimizer="CacheGlobalOptimizerFixedWeights"
        utility_function="CacheUtilityAggregateMin" max_capacity_kb="81920"
        watermark_capacity_kb="71680" compute_period_ms="500" purge_poll_period_ms="400"
        purge_on_insert="true" publish_stats_dataobject="true" keep_in_bloomfilter="false"
        handle_zero_size="true"  bloomfilter_remove_delay_ms="16000"
        manage_only_remote_files="true">
    <CacheKnapsackOptimizerGreedy />
    <CacheGlobalOptimizerFixedWeights min_utility_threshold="0.1">
      <Factor name="CacheUtilityAggregateMin1" weight="1" />
      <Factor name="CacheUtilityAggregateMax" weight="1" />
      <Factor name="CacheUtilityNewTimeImmunity" weight="1" />
      <Factor name="CacheUtilityAggregateSum" weight="1" />
      <Factor name="CacheUtilityRandom" weight=".1" />
      <Factor name="CacheUtilityPopularity" weight=".7" />
      <Factor name="CacheUtilityNOP" weight="0" />
      <Factor name="CacheUtilityNeighborhood" weight=".3" />
      <Factor name="CacheUtilityAggregateMin2" weight="1" />
      <Factor name="CacheUtilityPurgerRelTTL" weight="1" />
      <Factor name="CacheUtilityPurgerAbsTTL" weight="1" />
      <Factor name="CacheUtilityReplacementPriority" weight="1" />
    </CacheGlobalOptimizerFixedWeights>
    <CacheUtilityAggregateMin name="CacheUtilityAggregateMin1">
      <Factor name="CacheUtilityAggregateMax">
        <CacheUtilityAggregateMax>
          <Factor name="CacheUtilityAggregateSum">
            <CacheUtilityAggregateSum>
              <Factor name="CacheUtilityNeighborhood">
                <CacheUtilityNeighborhood discrete_probablistic="true" neighbor_fudge="1" />
              </Factor>
              <Factor name="CacheUtilityRandom"/>
              <Factor name="CacheUtilityPopularity">
                <CacheUtilityPopularity>
                  <EvictStrategyManager default="LRFU">
                    <EvictStrategy name="LRFU" className="LRFU" countType="VIRTUAL"
                          pValue="2.0" lambda=".01" />
                    <EvictStrategy name="LRU_K" className="LRU_K" countType="TIME"
                          kValue="2" />
                  </EvictStrategyManager>
                </CacheUtilityPopularity>
              </Factor>
              <Factor name="CacheUtilityNOP"/>
            </CacheUtilityAggregateSum>
          </Factor>
          <Factor name="CacheUtilityNewTimeImmunity">
            <CacheUtilityNewTimeImmunity TimeWindowInMS="12000" />
          </Factor>
        </CacheUtilityAggregateMax>
      </Factor>
      <Factor name="CacheUtilityAggregateMin">
```

```
            <CacheUtilityAggregateMin name="CacheUtilityAggregateMin2">
               <Factor name="CacheUtilityPurgerRelTTL">
                  <CacheUtilityPurgerRelTTL purge_type="purge_after_seconds" tag_field="ContentType"
                             tag_field_value="DelByRelTTL" min_db_time_seconds="1" />
               </Factor>
               <Factor name="CacheUtilityPurgerAbsTTL">
                  <CacheUtilityPurgerAbsTTL purge_type="purge_by_timestamp" tag_field="ContentType"
                             tag_field_value="DelByAbsTTL" min_db_time_seconds="1" />
               </Factor>
            </CacheUtilityAggregateMin>
         </Factor>
         <Factor name="CacheUtilityReplacementPriority">
            <CacheUtilityReplacementPriority>
               <CacheUtilityReplacement name="CacheUtilityReplacementTotalOrder" priority="2">
                  <CacheUtilityReplacementTotalOrder metric_field="MissionTimestamp"
                             id_field="ContentOrigin" tag_field="ContentType" tag_field_value="TotalOrder" />
               </CacheUtilityReplacement>
               <CacheUtilityReplacement name="CacheUtilityReplacementTotalOrder" priority="1">
                  <CacheUtilityReplacementTotalOrder metric_field="ContentCreationTime"
                             id_field="ContentOrigin" tag_field="ContentType" tag_field_value="TotalOrder" />
               </CacheUtilityReplacement>
            </CacheUtilityReplacementPriority>
         </Factor>
      </CacheUtilityAggregateMin>
   </CacheStrategyUtility>
</CacheStrategy>
```

The above configuration (extracted from within a Data Manager configuration) contains new parameters that are defined as follows:

<CacheStrategy>

name — Utility-based caching is implemented in ENCODERS as a cache strategy named "CacheStrategyUtility" under the <DataManager> tag.

<CacheStrategyUtility>

knapsack_optimizer — specifies which knapsack optimizer to use. Currently we only support the heuristic "CacheKnapsackOptimizerGreedy" optimizer.

global_optimizer — specifies the global optimizer which weights utility [-1,1] functions and thresholds in response to network feedback. Currently we only support the "CacheGlobalOptimizerFixedWeights" which uses fixed weights specified in config.xml.

utility_function — specifies which utility function to use for computing data object utility. Each utility function can be specified a name, which is referenced by the global optimizer. If no name is specified, the default is the utility function's class name.

max_capacity_kb — a hard constraint on the cache capacity. Data objects will be dropped immediately (without triggering the pipeline to make space) if their insertion will surpass this constraint.

watermark_capacity_kb — a soft constraint on the cache capacity. Data objects may be inserted causing this constraint to be violated. Upon execution of the pipeline, data objects will be evicted so that the watermark is not exceeded.

compute_period_ms — specifies a bound on the maximum frequency that a data object's utility is computed. Data objects will not have their utility computed more frequently than this period.

purge_poll_period_ms — specifies how often to run the pipeline. To disable poll based pipeline execution, set this value to 0.

purge_on_insert — "true" or "false". This enable/disables pipeline execution based on data object insertion. NOTE: event-based execution may be slow in systems with a large number of data objects.

publish_stats_dataobject — "true" or "false". This enable/disables publishing a data object containing cache statistics every time the pipeline is executed. Only the most recent statistics data object is kept in the cache. These data objects have an attribute "CacheStrategyUtility=stats". This should be used for debugging only.

manage_only_remote_files — "true" or "false". This enable/disables management of locally published files in addition to files received remotely. In most cases this value should be true, but false allows files published by an application to be deleted, which is possible, because Haggle takes ownership of those files. A value of "true" only manages files that are received remotely and are stored in the ~/.Haggle directory.

keep_in_bloomfilter — If true, then data objects evicted will remain in the bloomfilter, otherwise they are removed.

bloomfilter_remove_delay_ms — specifies time in milliseconds. keep_in_bloomfilter="false" is required. This option will wait the specified amount of time prior to removing the data object from the bloomfilter. It is useful to avoid strong assumptions on time synchronization in case of expiration-based purging.

discrete_size — specifies size in KB for discretizing the data objects to a less granular level when the knapsack optimizer computes the marginal utility. For example, if the discrete_size is 10KB then for all intents and purposes, the knapsack optimizer will treat a data object of 71KB and 72KB as the same size (and will then fall back to eviction based on create time if they have the same marginal utility). Default is 1 (acts the same way as before, no greater discretization).

Currently we support the following utility functions:

"CacheUtilityAggregateMin" : This utility function takes the minimum utility value of its constituent utility functions.

```
<Factor name="CacheUtilityAggregateMin">
   <CacheUtilityAggregateMin>
   ...
   </CacheUtilityAggregateMin>
</Factor>
```

"CacheUtilityAggregateMax" : This utility function takes the maximum utility value of its constituent utility functions.

```
<Factor name="CacheUtilityAggregateMax">
   <CacheUtilityAggregateMax>
   ...
   </CacheUtilityAggregateMax>
</Factor>
```

"CacheUtilityAggregateSum" : This utility function simply sums the values of its constituent utility functions, and ensures the compute utility is within [0,1].

```
<Factor name="CacheUtilityAggregateSum">
   <CacheUtilityAggregateSum>
   ...
   </CacheUtilityAggregateSum>
</Factor>
```

"CacheUtilityNeighborhood" : This utility function computes a utility based on the frequency of the data object within the 1-hop neighborhood (by inspecting the Bloom filter). This is usually used to reduce the utility of keeping a particular data object that frequently appears within a neighborhood, and the global optimizer multiplies the utility by a negative constant.

```
<Factor name="CacheUtilityNeighborhood" />
   <CacheUtilityNeighborhood discrete_probablistic="true" neighbor_fudge="0" />
</Factor>
```

The function examines the number of replicas R within the 1-hop neighborhood of size N to compute a probability of evicting the data object P as follows:
P = R / (N + F), where F specified in the *neighbor_fudge*.

discrete_probablistic — If true, then the utility will be 1 with probability P, and 0 otherwise. If false, then the utility is P.

"CacheUtilityNeighborhoodSocial" generalizes "CacheUtilityNeighborhood" by examining specific nodes within a social group (as defined by a common interest), possibly outside the 1-hop neighborhood.

```
<Factor name="CacheUtilityNeighborhoodSocial">
    <CacheUtilityNeighborhoodSocial discrete_probablistic="false" neighbor_fudge="4"
        default_value="0" social_identifier="squad" only_physical_neighbors="false" />
</Factor>
```

discrete_probablistic and *neighbor_fudge* are equivalent to the parameters of the same name in "CacheUtilityNeighborhood", and P is computed in the same way.

social_identifier — This defines the attribute A=(K,V) key K where V identifies the social group (nodes with a common interest).

only_physical_neighbors — If true, then only 1-hop neighbors within the social group will be inspected for replicas. If false, then all received node descriptions for the specified social group will be inspected.

default_value — This is the utility for a data object that is returned when the current node does not belong to any social group.

"CacheUtilityNeighborhoodOtherSocial" : This utility function is based upon the concept that nodes in the same social group will have a high likelihood of being in contact with nodes of the same group. This utility is biased towards the number of unique groups containing the content in question.

```
<Factor name="CacheUtilityNeighborhoodOtherSocial">
   <CacheUtilityNeighborhoodOtherSocial
    only_physical_neighbors="true"
    exp_num_neighbor_group="4"
    less_is_more="false"
    exclude_my_group="false"
    max_group_count="1" />
</Factor>
```

only_physical_neighbors — If true, looks only at the social group names of its 1-hop neighbors. If false, we look at all nodes in the known network nodes and use all their social group names.

exp_num_neighbor_group — Integer. This sets the expected number of social groups in mission. For example, if *exp_num_neighbor_group*=4, and a node only sees 3 distinct groups, it can assume there is a 4th, unseen group. This is handy for defined hierarchy social groups (e.g. military), where you know your own group, and how many other groups are out there.

less_is_more — If true, returns '(1-value)', otherwise, it returns '(value)'. This helps tweak the utility function to have more copies in more groups vs. high diversity w.r.t. unique data object.

exclude_my_group — If true, excludes a node's own group in the utility calculation.

max_group_count — Integer. This value sets the maximum number in a group to be counted, if they have the requested data object.

NodeManager configuration options can setup a node’s predefined group label to use the CacheUtilityNeighborhoodOtherSocial utility function. All nodes having the same label are considered in the same social group,e.g., <NodeManager social_group="Platoon1" >.

The relationships between nodes and group names are handled in NodeStore. All nodes not given an explicit social group name are considered to be in a single group.

CacheUtilityLocal — If true, ensures at least one copy of content is available in the network (on the publisher node). This is different from <CacheStrategyUtility manage_only_remote_files="true" /> option, which has the effect of not doing ANY processing on local files, which prevents replacement or time expiration.

```
<Factor name="CacheUtilityLocal">
   <CacheUtilityLocal protect_local="true" />
</Factor>
```

protect_local — If true, then all locally produced content will return a value of '1'. If false, or the content is non-local, it returns a '0'.

"CacheUtilityPopularity" uses LRU-K or LRFU to assign higher utility to data objects that are frequently accessed. We define an *access* as i) insertion, or ii) successfully sending the data object (DO) to a peer.

<EvictStrategy> defines the various LRU methods. Currently, we support 2 methods, LRU-k, and LRFU.
default — specifies the name of the default LRU.

All LRU modules are updated whenever a DO is received or a filter search matches the DO, but unless you specifically ask for a result by name, only the default LRU method result is returned.

For <LRU_K> or <LRFU>, if the name is not defined, it goes by the default type (e.g. <LRFU> default name is LRFU). countType is either COUNT (each DO that is accessed increments the count.

```
<Factor name="CacheUtilityPopularity">
<CacheUtilityPopularity>
   <EvictStrategyManager default="LRFU">
      <EvictStrategy name="LRFU" className="LRFU" countType="VIRTUAL" pValue="2.0"
lambda=".0000001" />
   </EvictStrategyManager>
</CacheUtilityPopularity>
</Factor>
```

CacheUtilityNewTimeImmunity — This utility function gives a positive value if the DO was received within a certain timeframe. Default is 2.5 seconds, but this can be adjusted with an xml option. Thus, all DO's that (by default) are within 2.5 seconds of the current time, are given a '1', and all that are not are given a '0'. This feature was added, as it was noted that large new DOs were dropped before they had a chance to disperse, while older DOs were distributed several times, giving them higher values.

```
<Factor name="CacheUtilityNewTimeImmunity">
   <CacheUtilityNewTimeImmunity TimeWindowInMS="120000" />
</Factor>
```

"CacheUtilityPurgerRelTTL" : This utility function computes a {1,0} utility based on whether the data object has expired by a relative received date. This utility function is a carry over from the CacheReplacement architecture. As in the previous architecture, it takes the parameters: "purge_type", "tag_field", "tag_field_value" and "min_db_time_seconds". Unlike previously, the DO is only evicted upon pipeline execution (as with every other utility function).

```
<Factor name="CacheUtilityPurgerRelTTL">
   <CacheUtilityPurgerRelTTL purge_type="purge_after_seconds" tag_field="ContentType"
tag_field_value="DelByRelTTL" min_db_time_seconds="1" />
</Factor>
```

"CacheUtilityPurgerAbsTTL" : This utility function is identical to the RelTTL, but uses absolute time.

```
<Factor name="CacheUtilityPurgerAbsTTL">
   <CacheUtilityPurgerAbsTTL purge_type="purge_by_timestamp" tag_field="ContentType"
tag_field_value="DelByAbsTTL" min_db_time_seconds="1" />
</Factor>
```

CacheUtilityNewTimeImmunity, CacheUtilityPurgerRelTTL, and CacheUtilityPurgerAbsTTL all have this configuration option:

linear_declining — If true, allows utilities to decline in a linear function (instead of a 1,0 utility).

```
<Factor name="CacheUtilityNewTimeImmunity">
   <CacheUtilityNewTimeImmunity TimeWindowInMS="10000" linear_declining="true" />
</Factor>
```

"CacheUtilityReplacementTotalOrder" : This utility function computes a {1,0} utility based on whether the DO is subsumed by another data object (using a total order replacement, specified in the configuration). This utility function is a carry over from the CacheReplacement architecture. As in the previous architecture, it takes the parameters: "metric_field", "id_field", "tag_field", and "tag_field_value".

```
<Factor name="CacheUtilityReplacementTotalOrder">
   <CacheUtilityReplacementTotalOrder metric_field="ContentCreateTime" id_field="ContentOrigin"
tag_field="ContentType" tag_field_value="TotalOrder" />
</Factor>
```

"CacheUtilityReplacementPriority" : This utility function computes a {1,0} utility based on an ordered list of replacement utility functions. Note that the specification is slightly different than aggregate utility functions (there are no <Factors> within the CacheUtilityReplacementPriority). This utility function is a carry over from the CacheReplacement architecture.

```
<Factor name="CacheUtilityReplacementPriority">
  <CacheUtilityReplacementPriority>
    <CacheUtilityReplacement name="CacheUtilityReplacementTotalOrder" priority="2">
      <CacheUtilityReplacementTotalOrder metric_field="MissionTimestamp" id_field="ContentOrigin"
tag_field="ContentType" tag_field_value="TotalOrder" />
    </CacheUtilityReplacement>
    <CacheUtilityReplacement name="CacheUtilityReplacementTotalOrder" priority="1">
    <CacheUtilityReplacementTotalOrder metric_field="ContentCreateTime" id_field="ContentOrigin"
tag_field="ContentType" tag_field_value="TotalOrder" />
    </CacheUtilityReplacement>
  </CacheUtilityReplacementPriority>
</Factor>
```

"CacheUtilityAttribute" : This utility function computes a utility based on a specific attribute. If Haggle receives a DO with attribute: utility=0.66, then it will assign a utility of 0.66 (it is still subject to the global optimizer and other utilities like every other function).

attr_max_value — specifies a factor by which to divide the utility, used to compute the utility over an attribute that isn't necessarily [0,1]. Default is 1.

```
<CacheGlobalOptimizerFixedWeights min_utility_threshold="0.34">
  <Factor name="CacheUtilityAttribute" weight="1" />
</CacheGlobalOptimizerFixedWeights>
<CacheUtilityAttribute attribute_name="utility" />
```

The following configuration parameters were used for data store optimization.

```
<DataManager ...>
 <DataStore>
  <MemoryDataStore
    count_node_descriptions="false"
    exclude_node_descriptions="true"
    in_memory_node_descriptions="true"
    exclude_zero_weight_attributes="true"
    max_nodes_to_match="30"
    shutdown_save="false"
    enable_compaction="false"
    run_self_tests="false" />
 </DataStore>
</DataManager>
```

Haggle *must* be started with the new `-m` parameter in order to use the No SQL database, and the MemoryDataStore option must be specified in the config.xml.

The following `MemoryDataStore` configuration options are supported:

count_node_descriptions — If true, system counts node descriptions when matching against *max_matches* parameter. Default is false.

shutdown_save — If true, system writes the database to disk during shutdown (uses SQLDataStore). Default is true.

exclude_node_descriptions — If true, system removes node description data objects from the result set when finding all DOs for a particular node description. Default is false.

in_memory_node_descriptions — If true, system does *not* keep node description data objects in the database (implies *exclude_node_descriptions*="true"`). Default is true.

exclude_zero_weight_attributes — If true, system does not use attributes with weight 0 in matching. Default is `false`.

max_nodes_to_match — Integer for the maximum number of nodes to match for a particular data object. Default is 30.

enable_compaction — If true, the database will be compacted upon sufficient deletions of nodes and data objects (compaction can be slow!). Default is false.

run_self_tests — If true, system runs the unit tests and report success/failure in the log file. Default is false.

The following configuration parameters were used for memory usage control:

```
<CacheStrategyUtility manage_db_purging="true" db_size_threshold="100000"
self_benchmark_test="true" >
```

The following options are currently supported:

manage_db_purging — If true, internal DB memory management is allowed.

db_size_threshold — (Value is of the form <long long>.) This option sets the watermark for internal DB DO entries. When this value is exceeded, the database will purge entries until the size is within this watermark.

self_benchmark_test — If true, sets up the self benchmark test. Normal operation cannot occur when this is set to true.

6 Configuration of Network Coding and Fragmentation

6.1 Network-coding Manager

The XML format that is accepted by the Network-coding Manager can be inferred from the example below, which is a part of a sample configuration file that enables the Network-coding Manager.

```
<NetworkCodingManager enable_network_coding="true"
            enable_forwarding="true" node_desc_update_on_reconstruction="true"
            max_age_decoder="300" max_age_block="300"
            resend_delay="0" resend_reconstructed_delay="10.0"
            delay_delete_networkcodedblocks="300.0"
            delay_delete_reconstructed_networkcodedblocks="10.0"
            min_network_coding_file_size="32769" block_size="32768"
            number_blocks_per_dataobject="1">
</NetworkCodingManager>
```

The parameters are defined as follows:

enable_network_coding — If false, disables network coding no matter the content or context.

enable_forwarding — *If* false, disables forwarding of network-coded blocks, which can reduce the routing overhead of network coding, but may make it more difficult for the receiver to reconstruct the content.

node_desc_update_on_reconstruction — If true, the system retransmits the updated node description (including the updated Bloom filter) of the receiver as soon as a data object has been reconstructed.

min_network_coding_file_size — specifies the minimum file size required before a data object is eligible for network coding

block_size — All network-coded blocks will have this size. Based on our experiments we recommend using a relatively large size such as 32K.

number_blocks_per_dataobject — specifies the number of blocks that will be included in a single network-coded data object. We recommend using one block per data object. Other settings are for experimental purposes only.

resend_delay — specifies the delay (in seconds) between network-coded blocks. This determines the rate at which a new blocks are generated and sent.

resend_reconstructed_delay — specifies the delay (in seconds) for sending (and hence reencoding) reconstructed data objects at intermediate hops.

delay_delete_networkcodedblocks — specifies the delay (in seconds) after which network-coded block data objects are deleted at the receiver and intermediate hops.

delay_delete_reconstructed_networkcodedblocks – specifies the delay (in seconds) after which network-coded block DOs are deleted at the receiver and intermediate hops if the corresponding data object has already been reconstructed. This should be smaller or equal to the previous (general) delay for deleting blocks.

max_age_block — specifies the maximum age of network-coded blocks (in seconds) in the caching layer at the encoder after the last send event. After this age, they are discarded.

max_age_decoder — specifies the maximum age of network decoder state.

decode_only_if_target – If true, only attempts to decode a data object if the current node is interested in the contents. This reduces memory consumption, but may lower delivery rates.

Certain nodes can be specified to perform network coding as long as the other conditions (such as *min_network_coding_file_size*) are met. Both *source_encoding_whitelist* and *target_encoding_whitelist* accept a comma-separated list of node names. Nodes matching the names in the lists are the only approved nodes, which can perform encoding (*source_encoding_whitelist*) and decoding (*target_encoding_whitelist*), respectively. Below is a typical configuration excerpt, which selectively enables network coding:

```
<NetworkCodingManager enable_network_coding="true"
        source_encoding_whitelist="n1,n2,n3" target_encoding_whitelist="n5,n7,n8"
        enable_forwarding="true" node_desc_update_on_reconstruction="true"
        max_age_decoder="300" max_age_block="300"
        resend_delay="0" resend_reconstructed_delay="10.0"
        delay_delete_networkcodedblocks="300.0"
        delay_delete_reconstructed_networkcodedblocks="10.0"
        min_network_coding_file_size="32769" block_size="32768"
        number_blocks_per_dataobject="1">
</NetworkCodingManager>
```

source_encoding_whitelist — is a list of comma separated values of node names that are allowed to encode content.

target_encoding_whitelist — is a list of comma separated values of node names that are allowed to decode content.

Network coding is computationally expensive so it is often useful to limit the encoding rates in addition to potential rate limits in the protocols. A corresponding delay is applied after potentially randomized delays defined in the Forwarding Manager and before delays specified in the protocols. For generality, we follow a similar pattern and allow the encoder delay to be a linear or quadratic function of the number of neighbors.

A typical excerpt with encoding delays may look as follows:

```
<NetworkCodingManager enable_network_coding="true"
        enable_forwarding="true" node_desc_update_on_reconstruction="true"
        max_age_decoder="300" max_age_block="300"
        min_encoder_delay_base="1000"
        min_encoder_delay_linear="10 min_encoder_delay_square="1
        resend_delay="0" resend_reconstructed_delay="10.0"
        delay_delete_networkcodedblocks="300.0"
        delay_delete_reconstructed_networkcodedblocks="10.0"
        min_network_coding_file_size="32769" block_size="32768"
        number_blocks_per_dataobject="1">
</NetworkCodingManager>
```

6.2 Fragmentation

A sample configuration file excerpt that uses fragmentation for all data objects larger that 1MB and network coding for the resulting fragments (if at least 32KB) is shown below. Network coding can be disabled if fragmentation is sufficient.

A sample configuration-file excerpt that uses fragmentation for all data objects larger that 1MB and network coding for the resulting fragments (if at least 32KB) is shown below. Network coding can be disabled if fragmentation is sufficient.

```
<FragmentationManager enable_fragmentation="true"
        enable_forwarding="true"
        node_desc_update_on_reconstruction="true"
        max_age_decoder="300"  max_age_fragment="300"
        resend_delay="0" resend_reconstructed_delay="60.0"
        delay_delete_fragments="300.0" delay_delete_reconstructed_fragments="60.0"
        min_fragmentation_file_size="1048577" fragment_size="1048576"
        number_fragments_per_dataobject="1">
</FragmentationManager>

<NetworkCodingManager enable_network_coding="true"
        enable_forwarding="true"
        node_desc_update_on_reconstruction="true"
        max_age_decoder="300" max_age_block="300"
        resend_delay="0" resend_reconstructed_delay="1.0"
        delay_delete_networkcodedblocks="300.0"
        delay_delete_reconstructed_networkcodedblocks="10.0"
        min_network_coding_file_size="32769" block_size="32768"
        number_blocks_per_dataobject="1">
</NetworkCodingManager>
```

The parameters of fragmentation are analogous to network coding and omitted here. It should be noted however that, unlike network coding, not all fragments have to be of equal size. The last fragment of a data object may be smaller that the specified fragment size.

6.3 Loss Estimation

The Loss Estimation Manager periodically assesses the loss rate of a link.
A sample configuration is listed below.

```
<LossEstimateManager>
        <PeriodicLossEstimate interval="5"/>
        <NetworkCodingTrigger loss_rate_threshold="0.1"
               window_size=”30.0” initial_loss_rate=”1.0”/>
</LossEstimateManager>

<FragmentationManager enable_fragmentation="true"
        enable_forwarding="true"
        node_desc_update_on_reconstruction="true"
        max_age_decoder="300"  max_age_fragment="300"
        resend_delay="0" resend_reconstructed_delay="60.0"
        delay_delete_fragments="300.0" delay_delete_reconstructed_fragments="60.0"
        min_fragmentation_file_size="1048577" fragment_size="1048576"
        number_fragments_per_dataobject="1">
</FragmentationManager>

<NetworkCodingManager enable_network_coding="false"
        enable_forwarding="true"
        node_desc_update_on_reconstruction="true"
        max_age_decoder="300" max_age_block="300"
        resend_delay="0" resend_reconstructed_delay="1.0"
        delay_delete_networkcodedblocks="300.0”
        delay_delete_reconstructed_networkcodedblocks="10.0"
        min_network_coding_file_size="32769" block_size="32768"
        min_time_between_toggles=”5.0”
        number_blocks_per_dataobject="1">
</NetworkCodingManager>
```

The above indicates that the loss rate is estimated every 5 seconds. The loss rate threshold for enabling network coding is 0.1, which means when the loss rate for a given link is higher than 0.1, network coding will be enabled. This setting is compatible with pure network coding or pure fragmentation. To use pure fragmentation, the loss rate threshold should be set to a value greater than 1.0. To use pure network coding, the loss rate threshold should be set to 0.0. The loss estimate is estimated over the last 30 seconds as specified by the *window_size* parameter. The *min_time_between_toggles* option in the NetworkCodingManager allows for a minimum delay between turning network coding on and off.

For LossEstimateManager to work, *enable_fragmentation must be true and enable_network_coding must be false*.

7 Configuration of Security

Integrity and Non-Repudiation. Because node descriptions are sent frequently, they can optionally be sent without signatures to reduce load, using the below entry in the node's configuration:

```
<SecurityManager sign_node_descriptions="false"/>
```

Signature verification of a data object requires that the receiver have the sender's public key. This may not hold if the two nodes have never met, or if the receiver does not trust the sender directly. To allow data objects to flow in such a case, we introduce *signature chaining*. This can be activated in a node's configuration with:

```
<SecurityManager signature_chaining="true" />
```

Decentralized Certification. We distinguish between nodes that are *authorities* versus those that are *users*. Any node can act as an authority, by adding an *Authority* section in the configuration.

```
<SecurityManager>
<Authority name = "CBMEN" />
</SecurityManager>
```

An authority node will listen to requests for certificate signatures and respond to them appropriately. Requests and responses are hashed and encrypted with a symmetric key that has previously been shared out-of-band between the authority and the user. Both authorities and regular nodes can specify shared secrets on a per-node basis, to allow different shared secrets to be used for communication with different nodes. Note that a shared secret must be the base64 encoding of a 128 bit key. The shared secrets can be specified in the configuration, as below:

```
<SecurityManager>
   <SharedSecrets>
      <Node id="ID2" shared_secret="N095WPD/lkSnFZIYefr9oQ==" />
      <Node id="ID3" shared_secret="D49nslmb7ph1j4xlXU6w5Q==" />
   </SharedSecrets>
</SecurityManager>
```

Any node can also specify the list of authorities that it trusts. It will request certificates from these authorities. Authorities can be declared in the configuration using the following configuration snippet:

```
<SecurityManager>
   <Authorities>
      <Authority id="ID1" name="CBMEN" />
   </Authorities>
</SecurityManager>
```

A node periodically sends out requests to get its certificate signature from the configured authorities, until it receives a signed certificate from each configured authority. The frequency of these responses can be configured as below:

```
<SecurityManager certificate_signing_first_request_delay=”2”
   certificate_signing_request_delay=”15”
   certificate_signing_request_retries=”-1” />
```

Confidentiality. When a node uses a policy for which it does not have a requisite encryption or decryption attribute, it issues a security data request. If the node is an authority, it handles such requests and sends appropriate responses. If the node does not receive a response, it will periodically repeat the request. The delay before attributes are requested can be configured. Encryption can be activated as shown below:

```
<SecurityManager encrypt_file_payload="true"    attribute_request_delay="30" />
```

Declarative Certification. An authority must specify which nodes it will authorize for certification. This can be specified in the configuration as below:

```
<SecurityManager>
<Authority name=" CBMEN ">
         <Node certify="true" id="ID2" />
         <Node certify="false" id="ID3" />
      </Authority>
</SecurityManager>
```

Any nodes that have the *certify* attribute set to *true* will have their certificate signature requests responded to with a signed certificate. Requests from other nodes will be dropped.

Declarative Authorization. An authority must specify which roles it will authorize for cryptographic attributes. Role names must be properly scoped by the authority’s own name. Each role can have a different set of associated cryptographic attributes for which it is authorized. For each role, a shared secret must also be specified in the SharedSecrets section. An example declaration of two roles is below:

For each attribute, a role may have access to the decryption key for the attribute, the encryption key for the attribute, or both.

To use roles, a *user* node must also specify role shared secrets for the roles it wishes to use. This configuration involves using exactly the same SharedSecrets section as in the previous snippet.

```
<SecurityManager>
   <SharedSecrets>
      <Role name="CBMEN.ROLE1"       shared_secret="9PbRR1ScJSgu1WW8IoMAuA==" />
      <Role name="CBMEN.ROLE2" shared_secret="1HPm/yvDNxzWtAMF7Y26yQ==" />
   </SharedSecrets>
   <Authority name="CBMEN">
      <Role name="CBMEN.ROLE1">
         <Attribute decryption="false" encryption="true" name="A1" />
         <Attribute decryption="false" encryption="false" name="A2" />
      </Role>
      <Role name="CBMEN.ROLE2">
         <Attribute decryption="true" encryption="false" name="A1" />
         <Attribute decryption="false" encryption="true" name="A2" />
      </Role>
   </Authority>
</SecurityManager>
```

System-level Configuration. Below is an excerpt from an authority configuration for a very high level of security (requiring that all data objects be signed and encrypted).

```
<SecurityManager attribute_request_delay="60" certificate_signing_first_request_delay="2"
certificate_signing_request_delay="15" certificate_signing_request_retries="-1"
charm_persistence_data="eJyrVkosLcnIL8osqYwvzlayUqiu1VFQKi1OLULiIpQUwMXSC0AspXQgqW
Role6aaxpq4unn7GribaGd61YZGWXkH1GRlpbtXJZXFlqkX+Fcbp6T7xpSZWHhGZCZ5RpkFpCabZo
UUO7ka5lRYJBXVFiV4eQSHFaemeur7+SfbmKUnlLuaqtUWwsACb01cg=="
encrypt_file_payload="false" max_outstanding_requests="40" security_level="VERYHIGH"
sign_node_descriptions="true" signature_chaining="false"
rsa_key_length=”512” composite_security_data_requests=”false”
temp_file_path="/tmp/haggletmpsecdata.XXXXXX">
   <SharedSecrets>
      <Node id="ID2" shared_secret="N095WPD/lkSnFZIYefr9oQ==" />
      <Node id="ID3" shared_secret="D49nsImb7ph1j4xlXU6w5Q==" />
      <Role name="CBMEN.ROLE1" shared_secret="9PbRR1ScJSgu1WW8IoMAuA==" />
      <Role name="CBMEN.ROLE2" shared_secret="1HPm/yvDNxzWtAMF7Y26yQ==" />
      </SharedSecrets>
   <Authorities />
   <Authority name="CBMEN ">
      <Node certify="true" id="ID2" />
      <Node certify="true" id="ID3" />
      <Role name="CBMEN.ROLE1">
         <Attribute decryption="false" encryption="true" name="A1" />
         <Attribute decryption="false" encryption="false" name="A2" />
      </Role>
      <Role name="CBMEN.ROLE2">
         <Attribute decryption="true" encryption="false" name="A1" />
         <Attribute decryption="false" encryption="true" name="A2" />
      </Role>
   </Authority>
</SecurityManager>
```

Next, we describe each parameter relevant to the Security Manager than an administrator might want to set in the configuration file.

security_level — specifies whether no data objects should be signed (LOW), only node descriptions should be signed (MEDIUM), all data objects should be signed (HIGH), or all data objects should be signed and encrypted (VERYHIGH).

signature_chaining — if true, a signature is added at every hop; otherwise, only the last hop’s signature will be present.

max_outstanding_requests — specifies the maximum number of SecurityDataRequests that can be enqueued. Limiting this number prevents excessive control traffic. The default is 40.

encrypt_file_payload — if true, content encryption is activated; otherwise, content is sent without encryption. This will always be set to true when the security level is VERYHIGH.

sign_node_descriptions — if true, node descriptions will be signed; otherwise, node descriptions will not be signed. This will always be set to true when the security level is HIGH or VERYHIGH.

certificate_signing_request_delay — specifies the amount of time in seconds before a request for an authority to certify this node's public key is sent again.

certificate_signing_request_retries — specifies the number of retries to send certificate signing requests. This is useful for testing purposes, the default is unlimited.

certificate_signing_first_request_delay — specifies the amount of time in seconds before the first request is sent to an authority to certify this node's public key.

attribute_request_delay — specifies the amount of time in seconds before a request for missing attributes is sent again.

charm_persistence_data — specifies the initial set of global parameters used by the CHARM cryptographic library. The same value must be set at all nodes in the system or the encryption and decryption functionality will not work correctly. A default value is set inside the Haggle binary, so this does not need to be set.

rsa_key_length — specifies the size of the RSA keys to be used for signatures. Higher values provide higher security, but generating larger certificates may take some time.

composite_security_data_requests — if true, "composite" security data requests are sent instead of requests for certificate signature; improving performance and saving round trips before receiving encryption keys.