Phase 1 CBMEN ENCODERS
Final Design Description
Version 1.0 - June 25, 2013

Submitted to:
Defense Advanced Research Projects Agency (DARPA), Strategic Technology Office (STO)
Attention: Dr. Keith Gremban, Phone: (703) 248-1547, Email: keith.gremban@darpa.mil

Submitted by:
SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025 USA
Attention: Dr. Mark-Oliver Stehr, Phone: (650) 859-5710, Email: mark-oliver.stehr@sri.com

Disclaimer: The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

Distribution Statement A: Approved for Public Release, Distribution Unlimited
Table of Contents

1 Introduction ......................................................................................................................... 3
  1.1 Program Objective ........................................................................................................... 3
  1.2 Value Proposition and Measures ..................................................................................... 4
  1.3 Purpose .......................................................................................................................... 5
  1.4 Scope .............................................................................................................................. 5

2 Phase 1 CBMEN ENCODERS Overall Design Description ................................................. 6
  2.1 Overall Architecture ....................................................................................................... 6
  2.2 Haggle as a Framework for CBMEN Development ......................................................... 7
  2.3 Mapping CBMEN Components into the Haggle Framework ........................................ 11

3 SRI Phase 1 CBMEN Software Description ....................................................................... 13
  3.1 Content and Metadata Distribution ............................................................................... 14
    3.1.1 Interest-Driven Content-Distribution Approach ....................................................... 14
    3.1.2 Initial Implementation and Haggle Integration ......................................................... 16
    3.1.3 Towards a General Architecture for Content Distribution .................................... 18
    3.1.4 Sample Configuration and Parameters .................................................................... 22
    3.1.5 Semantics of Haggle and First Class Applications ................................................... 32
    3.1.6 Basic Scenarios and Preliminary Results ................................................................. 37
    3.1.7 Limitations and Possible Future Directions ............................................................ 39

3.2 Content-based Protocols: Lightweight Unicast and Broadcast ....................................... 43
  3.2.1 Light-weight Unicast and Broadcast via UDP ............................................................ 44
  3.2.2 Towards a Content-Based Protocol Architecture for Haggle ................................... 45
  3.2.3 Sample Configuration and Parameters ..................................................................... 48
  3.2.4 Limitations and Possible Future Directions ............................................................. 53

3.3 Content-Based and Utility-Based Caching ...................................................................... 55
  3.3.1 Content-Based Caching Approach ............................................................................. 55
  3.3.2 Utility-Based Caching Framework ............................................................................. 58
  3.3.3 Applications of Content-Based Caching .................................................................... 61
  3.3.4 Towards a General Caching Strategy Architecture for Haggle ................................ 62
  3.3.5 Sample Configuration and Parameters ..................................................................... 67
  3.3.6 Limitations and Possible Future Directions ............................................................. 73
  3.3.7 Formalization of Total-Order Replacement ............................................................... 73

3.4 Interest Management and Adaptive Interest Modeling (AIM) ....................................... 76
  3.4.1 AIM Approach .......................................................................................................... 76
  3.4.2 AIM Inputs and Outputs ............................................................................................ 79
  3.4.3 AIM-Enabled Applications ......................................................................................... 82
  3.4.4 Interest Model Sharing .............................................................................................. 87
  3.4.5 AIM Implementation and Haggle Integration ............................................................. 88
  3.4.6 Sample Configuration and Parameters ..................................................................... 91
  3.4.7 Limitations and Possible Future Directions ............................................................. 95

3.5 Network Coding ............................................................................................................. 97
  3.5.1 Network Coding Approach ....................................................................................... 97
  3.5.2 Network Coding Implementation and Haggle Integration ...................................... 98
3.5.3  Sample Configuration and Parameters ............................................. 103
3.5.4  Limitations and Possible Future Directions ........................................ 105
3.6  Fragmentation .................................................................................. 107
   3.6.1  Fragmentation Implementation and Haggle Integration ...................... 107
   3.6.2  Sample Configuration and Parameters ........................................... 111
3.7  Security ......................................................................................... 112
   3.7.1  Security Approach ...................................................................... 112
   3.7.2  Using Access-Control Policies .................................................... 115
   3.7.3  Security Implementation and Haggle Integration ......................... 116
   3.7.4  Sample Configurations and Parameters ....................................... 120
   3.7.5  Limitations and Possible Future Directions ................................... 121
4  CBMEN Software Status and Schedule .................................................. 124
5  Summary & Conclusion ....................................................................... 125
6  Bibliography ....................................................................................... 127
1 Introduction

1.1 Program Objective

The objective of the DARPA’s Content-Based Mobile Edge Networking (CBMEN) program is to develop the network services and transport architectures to enable efficient, transparent distribution of content in mobile ad hoc networking (MANET) environments. CBMEN envisions application independent and network agnostic content distribution services that will be utilized by battlefield applications to efficiently distribute content. The goal of the CBMEN Program is to reduce latency and increase the effective throughput of content for warfighters at the tactical edge. The motivation for CBMEN is to significantly enhance the tactical effectiveness of the warfighter by providing the information he needs when he needs it. Note: security is also necessary such that information is provided only to those warfighters with authorized access and denied to everyone else, and the information should be delivered so that the receiver can verify the integrity and originator of the content (non-repudiation).

Tactical MANETs are subject to dynamic network topologies, network partitions, energy constraints, and wireless radio bandwidth limits. Current network communication is IP based and research into future Internet architectures has been exploring alternatives to IP due to its inherent limitations of its low-level of expressiveness and abstraction. The CBMEN ENCODERS (Edge Networking with Content-Oriented Declarative Enhanced Routing and Storage) is developing algorithms that operate at and exploit the higher-level of abstraction offered by a content-based networking architecture. With the CBMEN ENCODERS approach we are also advancing the state of art in content-centric networking by focusing on the storage and dissemination of content that is relevant in a given content, thereby further exploiting the richness of the available metadata and user/application interests.

To meet these technical objectives, the SRI International (SRI) team developed core technologies for content-based networking and provided support to SAIC, SRI’s selected mobile system integrator (MSI), who is responsible for integrating the technology developments from the technology developers (TDs) to produce functional content-based networks. In addition to SRI, the SRI team consists of SET (an SAIC company), Suns-Tech, and GPC (Gerla Phillips Consulting). Specifically, the SRI team includes Dr. Mark-Oliver Stehr (Principal Investigator and Technical Lead), Dr. Carolyn Talcott (co-PI), Dr. Minyoung Kim (Task Lead for Content Management), Dr. Ashish Gehani (Task Lead for Security), Dr. Ian Mason (Software Engineer), Tim McCarthy (Software Engineer), Chris Lockett (Technical Advisor) at SRI; Prof. J. J. Garcia-Luna-Aceves (PI), Prof. Hamid Sadjadpour, James Mathewson, Sam Wood, Dr. Srinivas Vutukury at Suns-Tech; Prof. Mario Gerla (PI), Josh Joy, and Yu-Ting Yu at GPC; and David Anhalt (PI), Ralph Costantini (Program Manager), Dr. Hua Li at SAIC/SET.

The SRI team has performed on two technology development tasks identified in the BAA:

- **Task 3**: Managing Distributed Content on a MANET system, and
- **Task 4**: Securing Content while Maintaining Accessibility.
To cover evaluation and integration activities the SRI team will perform on the following two additional tasks:

- **Task 5**: Self-Assessment and System Evaluation
- **Task 6**: Integration Support

Apart from building on results from earlier projects, such those funded in the context of the DARPA Disruption-Tolerant Networking (DTN) program and the current NSF- and ONR-funded projects on Networked Cyber-Physical Systems at SRI, the ENCODERS project has also leveraged some results and ideas developed by International Fellows in their own research while visiting SRI. Here we would like to use the opportunity to thank Hasnain Lakhani, Jong-Seok Choi, Dawood Tariq, Rizwan Asghar, Je-Min Kim, and Francoise Sailhan for their valuable ideas and contributions that if not already applied or translated to CBMEN have a lot of potential to influence the architecture and design choices in Phase 2 of the program.

We also would like to thank the SAIC MSI team (led by Dr. William Merrill and George Weston) for providing the development platform and successfully demonstrating an integrated CBMEN system based on the SRI ENCODERS architecture at Ft. AP Hill, VA, in May 2013, and MIT Lincoln Labs (the team led by Andrew Worthen) for their independent evaluation of the performance in testbed experiments and in the field.

It is important to note that in this Design Description document we focus on the design and implementation details of the CBMEN ENCODERS Phase 1 components developed by the SRI team and how they fit into the overall architecture. Before reading this this report, we recommend to take a look at our Final Phase 1 Report, which gives a high level introduction of the architecture and the ENCODERS functionality. The Final Report also contains the results of our performance evaluation for all ENCODERS components.

### 1.2 Value Proposition and Measures

Early on in the program Keith Gremban requested that we develop a Value Proposition for CBMEN and the measures needed to demonstrate the achievement of that Value Proposition. Based on the CBMEN Program Objectives, we proposed the following Value Proposition for CBMEN.

In the context of an edge tactical MANET (which includes: network disruption & reconnection; and limitations on bandwidth, range, transmission power, computing power, memory):

- Deliver the most relevant information available to the warfighter on a timely basis.
- Limit or eliminate the delivery of non-relevant information.
- Ensure that information is provided only to those personnel with authorized access, but without the inconvenience or overhead of key management.
- Provide additional layer of more fine-grained security and trust by operating on top of existing, possibly certified, security solutions.

The primary system performance measures against this value proposition are:

- **Accuracy:**
  - The percentage of “relevant information” that is delivered.
  - The percentage of “non-relevant” information that is delivered.
  These metrics are intended to measure the capability of CBMEN to accurately represent the users’ interest and use such representation effectively to deliver the right content.

- **Latency:**
  - For one-shot queries, the time from a query (given the information is in the system) to the time the relevant response is received.
  - For standing subscriptions the time from when the relevant information becomes available (given the interest is registered) until it is delivered to the interested party.

Note. Although accuracy and latency are the highest-level system performance parameters, they are by no means the only ones. The secondary performance parameters are measures such as bandwidth utilization, power consumption, computational power utilization, and memory utilization. These are all essentially measures of system efficiency. These measures are also important and are discussed in the referenced document. For further information see: *Value Proposition and Performance Measures for Content Based Mobile Networking (CBMEN)* by Ralph J. Costantini (SAIC).

1.3 **Purpose**

The purpose of this document is to describe the Phase 1 CBMEN ENCODERS (Edge Networking with Content-Oriented Declarative Enhanced Routing and Storage) design pursuant to meeting contract deliverable A0002.

1.4 **Scope**

CBMEN will be a two-phase program to develop the core algorithms and software for content distribution on mobile edge networks. The focus of this document is on the Phase 1 design. Where appropriate, references are made to Phase 2 capabilities; however the Phase 2 designs have not been completed and thus will not be described herein.
2 Phase 1 CBMEN ENCODERS Overall Design Description

2.1 Overall Architecture

A brief overview of the CBMEN architecture is provided in this section to facilitate an understanding of SRI’s Phase 1 Design. This overall architecture is depicted in Figure 1. Although the focus of this document is on the Phase 1 CBMEN capabilities, some of the capabilities depicted in the overall architecture are targeted for Phase 2. The Phase 2 items will be pointed out explicitly below. All other items are Phase 1.

Within each “box” in Figure 1, the organization responsible for that capability is indicated in parentheses and the boxes color-coded by organization as well – e.g., the SRI capabilities are in light blue, Drexel capabilities are in yellow, etc.

In the center of this figure is the Haggle framework with the enhancements to Haggle required to realize CBMEN functionality. This framework and its enhancements will be described a bit later in this document.

Above the Haggle framework API are several test applications and the Context-Aware Enrichment and Autogeneration Library (Drexel). The latter provides a rich meta-data API to the applications by coding rich meta-data as Haggle content of a special type. (Note: Haggle 0.4 provides attribute-value pairs but does not support complex logical
queries). In this way, CBMEN can support rich meta-data without re-working the existing Haggle infrastructure. A key benefit of this approach is that the rich meta-data can be distributed without having to distribute the associated content. This separation of meta-data from content allows the selective distribution of content based on interest matching and is a key feature supporting the efficient use of bandwidth / low latency of CBMEN since content is much larger in size than the meta-data that describes it.

Below the Haggle framework is the network stack. This includes a traditional UDP/TCP/IP network layer, an optionally multi-hop link layer (such as BATMAN-adv), as well as medium-access control and physical layers. Haggle 0.4 uses UDP to connect to local applications and TCP/IP to connect to peers. To support broadcasting to peers our MSI is currently investigating the extension of Haggle with an additional protocol module based on NORM (NACK-Oriented Reliable Multicast).

Next, a brief overview of Haggle will be provided in Section 2.2. This is followed by Section 2.3, which is an overview of the Haggle-based enhancements required to achieve the CBMEN capabilities.

2.2 Haggle as a Framework for CBMEN Development

A lynch pin of the SRI Team’s approach to the ENCODERS design is to leverage Version 0.4 of Haggle (more precisely we are using the slightly more recent development snapshot that was available at program inception, but the changes are minimal) as the starting framework for program development. Thus, an understanding of Haggle is essential to understanding the ENCODERS design.

In this section, a brief overview of Haggle is provided along with references to a more detailed description. The following was abstracted from (Erik Nordstrom, A Search-based Network Architecture for Mobile Devices 2009) and the Haggle WEB site (http://www.haggleproject.org/). Haggle is an open-source framework that was implemented and funded by the European Commission under the Information Society Technologies Program of the 6th Framework between January 2006 and June 2010.

Haggle is a search-based data dissemination framework designed for mobile opportunistic communication environments. This search-based approach is used for resolution (mapping data to interested receivers) and prioritization of sending and receiving data during encounters between nodes. Haggle provides underlying functionality for neighbor discovery, basic protocols, data object representation and storage, and basic interest resolution thus removing the need to implement such features in applications.

The Haggle architecture is depicted in Figure 2 below.
The Haggle architecture is event-driven, modular, and layer-less. These structural features provide flexibility and scalability. Central in the architecture is the kernel. It implements an event queue, over which managers that implement the functional logic of the architecture communicate. The kernel contains, apart from the event queue, a number of shared data structures, such as active neighbors, listening sockets, and also a data store that holds the relation graph. Figure 2 (left side) depicts how the kernel, managers and applications interact in the architecture. The circular structure of the architecture without any fixed ordering between the managers illustrates its layer-less design.

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 can delegate processing to modules that do work within their domains of responsibility. Modules are depicted in the figure as small circles attached to certain managers.

The Haggle code is relatively clean and well-structured but not optimized for performance, which will be a big challenge for CBMEN. Fortunately, the code is well commented and there is plenty of relevant documentation. Haggle is also multi-platform, and especially running on Linux, which significant speeds up the development cycle if used together with the CORE network emulation environment.
Even if performance can be improved, which is our objective for CBMEN, it should be noted that the Haggle architecture is intended for the communication of larger (i.e., semantically meaningful) chunks of information, e.g., files rather than packets, aiming at a higher time-scale than traditional packet switching networks, meaning that typical delivery times are in the order of seconds and minutes (possibly longer for pocket-switched networks) rather than milliseconds. The benefit of moving to higher timescale and to semantically meaningful units of information is the new level of expressiveness offered by content-based networking. Hence, the selection of the right applications is very important to exercise and demonstrate the benefits of our CBMEN solution.

**Subtleties of Haggle Version 0.4** Although there are several papers and technical reports describing Haggle (Erik Nordstrom, A Search-based Network Architecture for Mobile Devices 2009) (Franca Delmastro 2007) (Erik Nordstrom, Haggle: Relevance-Aware Content Sharing for Mobile Devices Using Search 2012), this description attempts to describe how Version 0.4 works in practice. There are some subtleties (especially with respect to node descriptions) that we emphasize here so that we can later accurately describe what needs to be modified to support the CBMEN architecture. Haggle applications express interest in content using a set of (key, value) attributes. Also, the application assigns a weight to each interest attribute that signifies the importance of the attribute. Each Haggle node supports multiple network interfaces without Haggle applications being made aware of the underlying network technology. For example, a Haggle node may concurrently communicate with one neighbor over Ethernet and another neighbor over Bluetooth, without a Haggle application writing custom code for each technology. Similarly, one could add a new interface module (say, for RS232) to the haggle kernel and existing Haggle applications will utilize it transparently.

A fundamental primitive in Haggle is the node description. Node descriptions describe the state of a Haggle node at a particular instant in time. It includes information such as: a list of all of the node’s interests, a summary of all of the data objects currently stored at the node (a Bloom filter), a list of all of the node’s network interfaces and their respective addresses, and a creation time stamp. Node descriptions are propagated throughout the network in human-readable XML format. By default, the Bloom filter is 2000 bytes and the entire node description requires 3 to 4 1500 byte packets, depending on whether the security module is enabled.

Periodically, each Haggle node issues a broadcast Hello packet on all of its interfaces to discover 1-hop neighbors. This is a UDP broadcast packet on all Ethernet interfaces. Upon two neighbors discovering each other, they will exchanges their node descriptions (see Figure 2 for typical use cases).

Logically, Haggle uses a single graph data structure (the data store) to perform the matching of content and interests (Erik Nordstrom, A Search-based Network Architecture for Mobile Devices 2009). Data objects and their associated attributes (metadata) are stored in this data structure. Additionally, Haggle treats node descriptions as data objects whose attributes represent the node’s interests. Upon
receiving a new node description or data object, Haggle will add it to the data store and trigger a resolution operation.

Whenever a new data object is inserted into the data store, Haggle performs a resolution operation over the data structure to build a list of all node descriptions in the data store that have an interest in common to the attribute specified by the data object. Haggle iterates across this list of node descriptions to forward the data object to the interested nodes. If a node description is that of the local machine, then the data is passed to the local Haggle application. If a node description is that of a 1-hop neighbor, then Haggle will send the data to the neighbor using the protocol appropriate for that neighbor’s interface. Otherwise, the node description is that of an interested node which is multiple hops away from the current node. This case triggers the delegate forwarding procedure, which is responsible for finding one or more 1-hop neighbors to forward the data to, as a relay to the interested node.

Data objects can be inserted in the data store either by a local Haggle application, or by receiving a data object from a neighbor. Whenever a new node description is inserted into the data store, Haggle performs an analogous operation where it examines the data store for all data objects that have an interest in common to the interest specified in the node description. Using the attribute weights and resolution parameters specified in the new node description, Haggle builds a ranked list of data objects that should be sent to the node represented by the node descriptions. As before, if the new node description is that of a 1-hop neighbor, then the matched data objects are forwarded directly using the corresponding protocol specified by the neighbor’s interface. Otherwise, the node description is that of a node which is multiple nodes away from the current node. This case triggers the delegate forwarding procedure, which is responsible for finding one or more 1-hop neighbors to forward the data to, as a relay for the interested node.

A key subtlety is that node descriptions are treated as data objects, where each node description’s set of interests is treated as attributes. This functionality is not fully explained in the Haggle documentation, but we verified that it occurs based on the code and an analysis of Wireshark traces. Specifically, resolution is slightly different as described above for newly inserted node descriptions: when a new node description is inserted into the data store and a list of data objects is generated to push to the new node, data objects containing node descriptions are included in this list. Similarly, this new node description is also pushed to any other nodes (whose node descriptions are in the data store) but only if there is a common interest.
2.3 Mapping CBMEN Components into the Haggle Framework

Although Haggle provides a robust framework, it serves only as a point of departure for the challenging capabilities required for CBMEN. These new CBMEN capabilities are realized via software implemented largely as new managers or new modules in the Haggle architecture. Now back to the center of Figure 1, which depicts the Haggle framework with the enhancements to Haggle required to realize CBMEN functionality. These Haggle-based enhancements, the CBMEN Tasks addressed and the organization accountable for their realization are given below. Note: Task 5, Self-Assessment and System Evaluation; and Task 6, Integration Support, are integral to all capabilities and are thus not explicitly called out below.

- **Drexel**
  - XML/RDF/OWL Representation of Rich Metadata and Queries (the raptor RDF library is used as a foundation)
  - Content Aware Enrichment and Auto-generation Library (a library that enriches the Haggle API without replacing it)
  - Registrar for Matching Rich Metadata and Rich Queries (implemented as a standalone registrar manager)

- **SRI**
  - Lightweight Dissemination of node descriptions and other small data objects such as rich metadata (Task 3, implemented as a new lightweight flooding module in the forwarding manager)
  - Interest-Driven Content Distribution (Task 3, implemented as a new routing module of the forwarding manager)
  - Cooperative Content-based Caching (Task 3, implemented as a module in the data manager)
  - Monitoring and Utility-Optimization (Task 3, part of caching in Phase 1, general framework with standalone manager in Phase 2)
  - Interest Modeling (implemented as standalone manager, supports caching in Phase 1 & 2 and anomaly detection in Phase 2)
  - Network Coding (Task 4, implemented as standalone manager)
  - Fragmentation as an alternative/complement to network coding and for comparison purposes (implemented as a standalone manager)
  - Content Signing and Policy-Based Encryption (Task 4, implemented as an extension of Haggle’s security manager)
  - Anomaly Detection (Task 4, Phase 2 only)

- **Harris**
  - Meta-data/Query Security (Harris has implemented an approach based on hashing)
The mapping of CBMEN components into the Haggle framework is illustrated in Figure 3. Note that the mapping has been slightly revised from our May 1, 2012 briefing, most notably caching is implemented as a new module in the data manager and network coding became a standalone manager for better modularity. Monitoring and optimization will be specialized to and integrated with caching (utility-based caching) in Phase 1, and will evolve into more general framework as part of the resource manager in Phase 2.
3 SRI Phase 1 CBMEN Software Description

For CBMEN Phase 1, the aforementioned tasks performed by the SRI team can be grouped into 5 work areas that extend the Haggle framework in orthogonal dimensions. A separate section in this design document will cover each of these work areas:

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. DIRECT exploits the most current state of network connectivity as opposed to PRoPHET (Anders Lindgren 2003) implemented in Haggle 0.4, which utilizes statistics about node encounters to predict the likelihood of connectivity in the future and hence targets pocket-switched networks rather than tactical MANETs. Both lightweight dissemination using flooding and DIRECT have been implemented as modules called from the forwarding manager as part of a new generalized content-distribution architecture.

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. Also, the expiration time and an optional delivery deadline, and more generally the utility of content is taken into account. In fact, 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 been marked with specific tags by the user, e.g., add a module which treats content for Blue-Force tracking differently than chat messages by discarding obsolete tracking data as early as possible in the network and as a by-product saving resources and making room for other important traffic. In the latest design, we have further generalized this notion using a utility-based caching framework, where utility of content can be defined as a function of multiple components.

3. **Adaptive Interest Modeling (AIM)** provides an accurate, real-time understanding of edge users’ dynamic interests and information needs that can be leveraged to provide efficient and effective content management by distributing only relevant content rather than flooding the network. The interest model is maintained by a new interest manager that has been added to Haggle. It can perform smart prefetching and will provide a notion of content utility that can be exploited by our caching algorithms. Based on a notion of similarity the adaptive interest model matches against a set of preloaded interest profiles that are in turn activated to prefetch relevant content. As an extension, interest profiles can also
be published by other nodes (e.g., a commander or a squad leader) to establish common interest and to prefetch (and hence to proactively replicate) information relevant to a given mission. In the latest version, interest models can be shared between nodes and hence collectively learned and used.

4. **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 also 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. In addition to network coding we have implemented fragmentation, which can be used as an alternative to network coding and as a baseline to evaluate network-coding performance. More interestingly, it can be used in combination with network to reduce network-coding overhead.

5. **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. The complexity is hidden from the user who simply tags published content with a suitable access policy attribute.

3.1 **Content and Metadata Distribution**

This section describes a mechanism that enables much more flexible content caching than is provided by Haggle 0.4. At a high-level, Haggle is an application-layer (although it supports a variety of layer 2, 3, and 4 technologies) content-centric framework for pocket switched networks (Anders Lindgren, 2003). Applications running on Haggle nodes express interests in particular types of content, and they also add content to the network with metadata. The role of Haggle is to efficiently send content in the network to interested nodes. In other words, Haggle matches content to interests. To this aim, Haggle must disseminate interests of nodes throughout the network, and then disseminate the relevant content to interested nodes.

3.1.1 **Interest-Driven Content-Distribution Approach**

The first step in investigating content caching is to examine content dissemination and discovery. Unmodified Haggle can support the following methods of content and interest dissemination, either explicitly or implicitly by assigning interests to certain nodes: 1) epidemic dissemination (flooding) of content and interests (all node interests and content share a common attribute); 2) single-hop dissemination (no routing) of interests and content (disable delegate forwarding); and 3) multi-hop encounter based
dissemination of interests and content (PRoPHET (Anders Lindgren, 2003) delegate forwarding).

Our approach is inspired by DIRECT (Disruption REsilient Content Transport), a content dissemination protocol for disruption tolerant networks (Ignacio Solis, 2008). DIRECT offers an alternative to these three approaches. DIRECT disseminates interests epidemically, but unicasts content from source(s) to the destination. Presumably, interests are smaller in size than data, so this approach may utilize network resources more efficiently than an approach which floods content. Similarly, a DIRECT implementation will enable a side-by-side comparison with PRoPHET, an encounter-history-based approach.

As in Haggle, in DIRECT nodes publish objects and assign attribute metadata to each object. As in Haggle, if a node wishes to receive some piece of content it issues a query that expresses the interest in terms of attributes. Unlike in Haggle, DIRECT offers a different set of primitives for both disseminating and retrieving content. DIRECT supports HANDLE and SEND primitives, where HANDLE allows objects to be generated on-demand in response to a query, and SEND allows a sender to push content directly to a known receiver.

DIRECT also supports 3 different types of queries: pub-match, all-match, and stop-match. In pub-match, the query will only be answered by the publisher. In all-match, every object that matches the query will be sent back. In stop-match, the first object that matches will be returned and the query is not re-broadcast. Note that there is no guarantee that only one data object will be returned to the query initiator in a stop-match. In our initial design we have implemented all-match, because this is closest to the semantics of Haggle with standing subscriptions. However, we also see an opportunity to exploit the stop-match mode in Haggle, e.g., to access data objects based on their unique identifier potentially with lower overhead. Stop-match may also be of interest in the future to optimize certain one-shot queries for which only a single result is expected or needed.

DIRECT uses flooding to propagate the interests, where the initiator increments the sequence number of the query and initiates another flood in the event that no data is returned. Also, each node maintains a query table to keep track of which neighbor it first received a particular query from. Upon receiving data that matches a query, the node will store the object in its data table and forward the data to the neighbor specified in the query table. If a data object is forwarded to a node that is not interested in the data, and if it does not have an entry in the query table, then it advertises the data object to its 1-hop neighbors so that they can request it.

Nodes discover each other using periodic hello messages to maintain a neighbor table. If a node hears a 1-hop neighbor, then it marks it as active. If a node no longer hears a neighbor after a period of time, then it marks the neighbor as remote. When a node is marked active, the two nodes exchange query information to propagate queries that have not been answered yet.
3.1.2 Initial Implementation and Haggle Integration

We wish to integrate DIRECT in Haggle in a way that leverages the functionality already provided by Haggle, especially the fact that in Haggle each node maintains knowledge about other nodes and the content they have already received and cached. Additionally, our approach should be extensible and fast to implement, to rapidly prototype different variations of DIRECT and quickly uncover unforeseen problems. To support a large class of scenarios and improve performance, we also wish to provide an extensible mechanism of distributing content proactively, without explicit user subscriptions.

To evaluate DIRECT’s effectiveness in the context of CBMEN, we have implemented several revisions of a preliminary version of DIRECT: Alpha-DIRECT and Alpha-DIRECT++. So far all of these versions support all-match semantics, and they are implemented as a delegate forwarders, with slight modifications to the forwarding manager. We observe that DIRECT has two phases: 1) the interest flood, and 2) reverse path data forwarding. Haggle already provides a mechanism for distributing node descriptions which contain an interest list, based on interest commonality. Below, we describe the general approach to our integration of DIRECT and Haggle.

For Alpha-DIRECT (the first iteration of DIRECT) we exploited the inherent one-hop propagation mechanism supported by Haggle, by making sure that all nodes share a common interest attribute, such as key = “NodeDescription”, value = “”, weight = 1, if we want to reuse an abstract version of Haggle’s native node description attribute (which in unmodified Haggle has the node identifier as its value). When a node’s interest changes, it will trigger a flood of its node description (and consequently its interests) to all nodes within a connected component (as DIRECT does when a node floods an interest). In this version, when a node’s interest list changes or a new data object is added its node description it is re-flooded (in the same way DIRECT sends a flood when it has a new interest, but without propagation across connected components). Haggle’s use of Bloom filters prevents an already received node description from being retransmitted.

To address reverse path data forwarding, Haggle must remember from which neighbor it received a node description (and consequently, the node’s interest). Whenever Haggle receives a data object, it records the remote interface from which it received the object. Thus, we do not need to add another data structure (i.e. an interest direction table) to keep track of where the interest was heard from; we can inspect the existing node store to find the delegate (one-hop neighbor) node that belongs to the remote interface for the target. Similarly, when generating a list of targets for a particular delegate, we can use the data store and node store to find all the node descriptions that were received through that neighbor. Note that this approach reuses the aging and cleaning mechanisms already in place for the data store, whereas if we added another data structure we would need to implement an aging mechanism that maintains consistency with the data store.

To implement this logic, we added a new delegate forwarder module (ForwarderAlphaDirect.cpp) that can be enabled or disabled through the configuration
file. Whenever Haggle forwards content over multiple hops, Haggle calls into the ForwarderAlphaDirect module which inspects the data store and node store to generate the appropriate delegate for a target, or the appropriate targets for a delegate. In the case where the ForwarderAlphaDirect selects a delegate, which is no longer a neighbor (perhaps the neighbor moved away) forwarding fails. It is conceivable that a future version could have a configuration option that will revert to a different delegate forwarder (i.e., PRoPHET) for hybrid delegate forwarding.

**Limitations of Alpha-DIRECT**

Alpha-DIRECT served as a bare-bones prototype of a combined DIRECT and Haggle implementation, however it has limitations both in its implementation and in its design. With respect to the implementation, this prototype requires the use of a shared attribute for all of the nodes in the network in order for node descriptions to propagate (which contain a node’s interest and Bloom filter). This trick makes testing more difficult, since an application must explicitly register a hard-coded interest. More generally, the shared attribute has unclear semantics when other applications also have an interest in the attribute. Similarly, if a node has set a maximum number of matches, then potentially the hard-coded attribute can change the existing Haggle semantics: some files may or may not be transferred when DIRECT is enabled or disabled.

With respect to the design, our initial version of DIRECT did not refresh or explicitly timeout existing node descriptions. Traditional DIRECT uses a soft-state approach of re-flooding interests in order to find new paths and maintain existing paths. This re-flooding is based on a timer (20 seconds in the original paper). In contrast, Alpha-DIRECT’s use of a shared attribute will only flood a node’s node description when its interest changes. Thus, our initial version of DIRECT performed poorly with the network topology changes, since alternative paths were not discovered automatically.

In response to these limitations, we developed Alpha-DIRECT++. Here, we added periodic node description refresh, and node description purge. We also optimized Alpha-DIRECT to make fewer calls to the database when disseminating node descriptions, by adding an in-memory node store cache which contains all discovered node descriptions (not just one-hop neighbors).

**Limitations of Alpha-DIRECT++**

Although Alpha-DIRECT++ can support a variety of scenarios, it is still relying on the publisher to receive an explicit interest from the sender before disseminating content. To support a larger class of scenarios, we expanded upon the flooding mechanism primarily used for node description dissemination, to support a basic form of proactive replication.

Another limitation of Alpha-DIRECT++ was the use of the data object abstraction as the fundamental unit of content dissemination. Thus, Alpha-DIRECT++ would transmit the file at a hop-by-hop basis, as opposed to breaking the files into fragments and routing
each fragment, which occurs in traditional routing protocols. To address this limitation, we have successfully integrated fragmentation and network coding with DIRECT. This combination offers greater content dissemination performance advantages in networks with unreliable, short-lived paths.

3.1.3 Towards a General Architecture for Content Distribution

We refactored and generalized Haggle’s forwarding architecture and extended our initial DIRECT design. Inspired by our earlier work on interest-driven routing in the DARPA DTN program (Mark-Oliver Stehr, 2008), we want to distinguish lightweight knowledge about the network and application content and use different dissemination mechanisms for each of these classes. We have taken this idea one step further and extended the forwarding manager to allow the user to configure a content classifier module and one or more forwarder modules that can register for a particular content class. As an instructive example of content classification, our basic classifier assigns one of two possible tags, depending on whether the content is “light weight” or “heavy weight.” It assigns the light-weight tag for node descriptions, and the heavy-weight tag for all other data objects. This use of such classifiers enables Haggle to choose forwarding protocols dynamically based on the content that is to be forwarded. In the current version, there must be exactly one forwarder module for each content class or a default forwarder that is used for content not covered by the classifier. Note that classifiers can be arbitrarily composed, using the priority classifier, to form more complex rules.

ForwarderAlphaDirect and ForwarderProphet are examples of forwarding modules that are typically registered for heavy-weight content. This registration is defined in the configuration file. We also added a ForwarderFlood that is typically registered for light-weight content. This forwarder forwards content to all of the node’s neighbors, and it relies upon the Bloom filters to avoid routing loops and redundant transmissions (using the same method that our initial version of DIRECT uses, but without the need for shared attributes).

**Message Ferrying** As another generalization of DIRECT, we support two different types of flooding for node descriptions and arbitrary data objects: 1) contemporaneous flooding without push on contact, and 2) contemporaneous flooding with push on contact. Contemporaneous flooding simply refers to the method of immediately forwarding a data object that is marked for flooding to each of a node’s neighbors (respecting the Bloom filters). Push on contact refers to a method of triggering subsequent contemporaneous floods upon discovery of a new neighbor. Specifically, when this option is enabled and a new neighbor is discovered, the discovering node will push all of the data objects marked for flooding to the newly discovered neighbor (which may trigger an additional flood to its neighbors). Push on contact is essential for supporting message ferrying scenarios, where the message ferry is not attached to the connected component of the data publisher at the time of publication, and the subscriber is in a different connected component of the publisher.

**Proactive Replication** Note that the generalization of ForwarderFlood to support flooding of arbitrary data objects is a form of proactive content replication: data objects are
proactively pushed to nodes which may not have any interest in the data, for the sake of increasing the delivery ratio, or reducing the delivery latency. This feature can deliver content (albeit at the expense of an increase in overhead) of content in scenarios where DIRECT and Prophet cannot, for example in one-way message ferrying.

To enable the periodic dissemination of interests and Bloom filters, we extended the node manager to read a configuration parameter that specifies a node description refresh period and jitter. With these two parameters, nodes can periodically broadcast their own node descriptions even if their state is unchanged. The refresh period is a measurement in milliseconds that indicates how frequently the node description should be sent, and the jitter is a measurement in milliseconds of how much random time should be added to the period, in order to avoid synchronization problems. Specifically, we will draw a measurement in milliseconds uniformly at random from the interval $[0, \text{jitter})$ and add this to the period to determine when to broadcast the next node description.

Additionally, we updated the node manager to purge node descriptions that are older than a maximum node description age (based on creation time). This age is specified as a configuration parameter. This purging prevents Haggle from maintaining unnecessary state and routing on paths that are no longer valid.

To avoid frequent database accesses due to node description propagation, we added an in-memory cache that sits directly above the database and is used solely for storing node descriptions. Note that node descriptions that are not within the immediate neighborhood of a particular node are also stored in this cache. This is a generalization over unmodified Haggle, which cached only neighbors in the node store.
Existing Classes

ManagerModule: All modules in Haggle descend from this class. It provides a thread that is associated with a specific Haggle manager, and a simple interface for configuration handling. Derived classes can retrieve the associated manager and post events to the main event queue.

Modified Classes

Forwarder: All forwarder modules in Haggle descend from this abstract class. It provides an interface for the forwarding manager, so that the module can learn about the current network state and make forwarding decisions. Specifically, the Forwarding Manager will defer forwarding decisions to a Forwarder Module whenever delegate forwarding occurs. Recall that delegate forwarding occurs when a data object matches a node description, a target, that is not an immediate neighbor: Haggle must find a suitable set of neighbors to forward the data object to in order to reach the target.

ForwarderAsynchronous: This abstract forwarder module serves as a base class for other forwarder modules that need asynchronous functionality. It implements an event queue that serializes callbacks using forwarder tasks. Forwarder tasks do work on behalf of forwarders that derive from ForwarderAsynchronousInterface, enabling
multiple ForwarderAsynchronousInterface forwarders to share the same event queue (and serialize their work) and thread.

**ForwarderAsynchronousInterface:** This abstract class provides an interface for forwarders that need asynchronous functionality. Forwarders that derive from this class associate with a single ForwarderAsynchronous, by passing a ForwarderAsynchronous object to the constructor. This class overrides all of the Forwarder methods to post events on the task queue from ForwarderAsynchronous, as opposed to blocking and directly executing the respective methods. In Haggle 0.4 ForwarderAsynchronous included this interface, so strictly speaking ForwarderAsynchronousInterface is a new class.

**ForwarderProphet:** This concrete forwarder module is asynchronous (derives from ForwarderAsynchronousInterface) and it is typically used to forward “heavy weight” data objects. It maintains a probabilistic encounter history table and uses aging to decide to which neighbor to forward a specific data object.

**New Classes**

**ForwarderAlphaDirect:** This concrete forwarder module is asynchronous (derives from ForwarderAsynchronousInterface) and it is typically used to forward “heavyweight” data objects. It is based on the DIRECT protocol, but adapted to fit within the Haggle framework. A node uses this class to forward a data object for a target to the neighbor that first “told” the node about the target.

**ForwarderFlood:** This forwarder enables flooding of both node descriptions and arbitrary data objects. It supports two modes of flooding: 1) contemporaneous flooding without “push on contact”, and contemporaneous flooding with “push on contact.” Push on contact allows newly discovered nodes to receive data objects that were previously flooded. In this case, the new node floods the data object within the expanded connected component. The same Bloom filter mechanism is used to prevent redundant transmissions, as is used with any data object transfer in Haggle.

**ForwardingClassifier:** This abstract module is responsible for analyzing data objects and “tagging” them according to some criteria. Forwarder modules can be registered with a specific tag, and other modules use the classifier to lookup the associated forwarder for a specific data object.

**ForwardingClassifierBasic:** This concrete classifier module tags content in two categories: “light weight” and “heavy weight”. Specifically, node description data objects are tagged as lightweight and all other data objects are tagged as heavyweight. These tags are specified in the configuration file.

**ForwardingClassifierNodeDescription:** This concrete classifier module tags node description data objects with a specified tag. This tag is specified in the configuration file.
ForwardingClassifierAttribute: This concrete classifier module tags content that has a specific attribute. Both the attribute and the tag are specified in the configuration file.

ForwardingClassifierSizeRange: This concrete classifier module tags content that is within a customizable size range. Both the range and the tag are specified in the configuration file.

ForwardingClassifierAllMatch: This concrete classifier module tags all content with a custom tag. The tag is specified in the configuration file. This classifier is useful for debugging and when used with the priority classifier.

ForwardingClassifierPriority: This meta-classifier allows the user to compose multiple classifiers according to a total order. Specifically, it resolves conflicts, which may occur when two classifiers both have a valid tag for a single data object, by selecting the relevant classifier with the highest priority.

ForwarderAggregate: This concrete forwarder module is an asynchronous forwarder that contains multiple forwarders. For network state updates it proxies the information to all of the contained forwarders. For delegate specific methods, it uses the classifier to lookup the specific forwarder for the delegated data object, and passes the data object to the responsible forwarder. For example, one could use the combination of ForwardingClassifierBasic, ForwarderFlood and ForwarderAlphaDirect modules to flood all node descriptions (light-weight class) and forward all application content (heavy-weight class) using AlphaDirect.

ForwardingClassifierFactory: This factory class is responsible for instantiating and configuring classifier objects given a specific classifier name.

ForwarderFactory: This factory class is responsible for instantiating and configuring forwarder objects according to a configuration object. Depending on the passed configuration object, the factory may also instantiate a classifier (using the ForwardingClassifierFactory) to construct a ForwarderAggregate.

3.1.4 Sample Configuration and Parameters

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.

Below is an example excerpt from the configuration file that enables light-weight flooding of node descriptions, proactive replication of small data objects and tagged data objects, whereas it uses DIRECT for specifically tagged data objects and for all other data objects not covered by the previous cases. Note that 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">
    <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>
  <Forwarder protocol="Flood" contentTag="lw">
    <Flood push_on_contact="true" />
  </Forwarder>
  <Forwarder protocol="AlphaDirect" contentTag="hw" />
</ForwardingManager>

ForwardingManager

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 original Haggle, but we recommend to set 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:

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

A typical except that activates Prophet looks as follows:

```xml
<ForwardManager query_on_new_dataobject="true" periodic_dataobject_query_interval="0"
                    recursive_routing_updates="false" periodic_routing_update_interval="10">
  <Forw der 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>
</ForwardManager>
```

NodeManager

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

```xml
<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>
```

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. This prevents DIRECT from forwarding on invalid paths (paths which 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.

Below is an excerpt from `config.xml` that demonstrates how to enable this optimization. Omitting this parameter disables this optimization. We recommend always enable this optimization (except in small networks for testing/debugging purposes).

```xml
<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 place node descriptions in the database. Note that this optimization should only be used in conjunction with the forwarder flood mechanism for light weight content, since 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 set them for "this" node. This replaces potentially invalid state in the stale in the non-counting Bloom filter.

In the latest version, 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 set to 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.

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, which enables the in-memory database without journaling:

```xml
<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:

```xml
<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>
```
In unmodified Haggle, each node description has a

\[\text{NodeDescription}=<\text{nodeid}>\]

attribute (with weight 1), which is used by Haggle internally to detect node descriptions. By abstracting from the specific \(<\text{nodeid}>\), using the setting \text{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 a sample for 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 \text{node_description_weight}="0" to prevent node descriptions to interfere with the application semantics. This is useful together with the first-class application feature.

The NodeDescription attribute is used when node descriptions are stored in the database like other objects (as in unmodified Haggle) but it is not needed when they are kept in memory as possible in our new routing framework (see \text{in_memory_node_descriptions}). To this end, we recommend to eliminate this attribute in order to save bandwidth using \text{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>
```

**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:

```xml
<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 (used by the Drexel registrar manager). 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.

### 3.1.5 Semantics of Haggle and First Class Applications

The mathematical semantics of Haggle query resolution is defined in (Erik Nordstrom, A Search-based Network Architecture for Mobile Devices 2009). In a nutshell, the relevance of a data object relative to a weighted set of interests (e.g., a node description representing the query) is computed by the weighted sum of overlapping attributes. Weights always refer to weights of interest attributes, not attributes of data objects (which are not weighted). The result of a query is the ranked set of data objects up to a given matching threshold and up to a certain maximum rank (limiting the number of results in the query). Weights and thresholds in Haggle are represented as integers in the range 0...100.

This is a fairly general and powerful concept that goes beyond traditional disjunctive and conjunctive queries. Unfortunately, Haggle does not implement this semantics consistently. For instance, the attributes of node descriptions are aggregated from multiple applications an additional internal attribute is added so that from the viewpoint of the application the semantics looks quite different. Another complication is that Haggle uses two different variations of matching, one for applications (called filtering) and one for routing (called matching). Our experience is that in unmodified Haggle only a disjunctive interpretation of interest (that is consistent with aggregation by union) is of practical use.

In the ENCODERS project we have modified Haggle to consistently use the above-mentioned mathematical semantics at the application-level, which is what matters for end-users. In this version application nodes become first-class citizen (as opposed to be approximately represented by their proxy, the device node). Aggregation is disabled in favor of accuracy.

This feature is called **first-class applications**, and can be enabled by turning on the Boolean first_class_applications config option of the application manager, as in the following excerpt:
Turning the feature off (the default setting) should yield the unmodified Haggle semantics with aggregation, but it is not recommended, because our routing algorithms can be expected to exploit the first-class application semantics (especially the separation of application interest and device Bloom filters that comes with it) in the future.

It should be noted that the first-class application feature nicely works together with our extended Haggle API functions

\[
\text{haggle_ipc_set_matching_threshold and haggle_ipc_set_max_data_objects_in_match,}
\]

which allows applications to set the above-mentioned parameters (threshold and maximal rank) for each application independently (which was not possible in unmodified Haggle). With these functions it is possible to support not only fully general threshold queries, but also disjunctions of conjunctions (i.e., precise logical queries).

The default settings for the above mentioned parameters are 0 and 100, respectively, and can be configured in the configuration file, e.g. using

\[
\text{<NodeManager>}
\text{ <Node matching_threshold="0" max_dataobjects_in_match="100"/>}
\text{ </NodeManager>}
\]

Unmodified Haggle supports the use of wildcards for matching in the local database, but not for remote searches. With the first-class applications features, there is a single semantics for both kinds of matching, and wildcards are currently not supported.

The following tutorial uses our test program, so-called “hagletest”, to exercise the extended functionality of the new semantics. All these tests require the following configuration setting, which enables Haggle to maintain first-class applications, i.e., explicit representations of all subscriptions (i.e., application node descriptions) in the network:

\[
\text{<ApplicationManager first_class_applications="true">}
\text{ </ApplicationManager>}
\]

To make sure that the NodeDescription attribute does not interfere with the semantics, make sure to set node_description_attribute_weight="0" and, if you are not using in-memory-node descriptions, better set count_node_descriptions="false", as in the following excepts:
NOTE: It is instructive to use an SQLite browser to inspect the state of the Haggle database haggle.db while executing the following steps.

**Basic Queries**

Publish data objects with two attribute/value pairs:

```
haggletest pub ContentType=Map Location=Kirkuk
```

The data object can be retrieved by the following subscription:

```
haggletest -z -c sub Location=Kirkuk
```

The option -z clear the application Bloomfilter after registration so that potentially delivered objects can be delivered again, and -c clears the application interest to start with a well-defined state.

**Threshold Queries**

You may execute pub and sub on the same or on different nodes. The behavior should be equivalent.

Publish data object with three attributes:

```
haggletest pub A B C
```

Disjunctive query:
Query for all data objects with attribute X or A.

```
haggletest -z -c nop
haggletest sub X A
```

After subscribing the published data object should be returned.
(You can exit haggletest with Ctrl-C).
Conjunctive query:
Query for all data objects with attribute X and A.

haggletest -z -c nop
haggletest -t 100 sub X A

The option -t 100 means that a threshold of 100% needs to be reached to count as a match, i.e., all attributes of the subscription need to be included in the data object we are looking for.

No data object should be returned in this case.

Conjunctive query:
Query for all data objects with attributes A, B, and C.

haggletest -z -c nop
haggletest -t 100 sub A B C

The published data object should be returned.

N-out-of-M query:
Query for all data objects with at least 66% of attributes A, B, and X, meaning at least two out of the given three must be present.

haggletest -z -t 66 sub A B X

The published data object should be returned.

In contrast

haggletest -z -t 66 sub A X Y
one matching attribute is not sufficient for a data object match.

Threshold query:
Query for all data objects with at least 80% relevance given weighted attributes A,B,C,D with 50%,30%,10%, and 10% importance, respectively.

haggletest -z -c nop
haggletest -t 80 sub A:50 B:30 C:10 D:10

The published data object should be returned.

haggletest -z -c nop
haggletest -t 80 sub A:10 B:10 C:30 D:50

No data object should be returned with the query.
**Limiting the Number of Matches**

You may execute pub and sub on the same or on different nodes. The behavior will NOT be equivalent.

Publish 100 data objects with different time stamp and sequence number attributes, and an attribute called TEST.

```bash
haggletest -b 100 pub TEST
```

Query for data objects with TEST attribute limiting number of matches on remote nodes to 10.

```bash
haggletest -c nop
haggletest -m 10 sub TEST
```

(-m 10 can be omitted if it is configured are default)

All 100 data objects should be returned if sub is executed on the same node as pub.

Otherwise, the number of data objects returned is limited to 10. In that case the next 10 objects can be obtained by using another invocation (without -z, otherwise the same data objects are returned):

```bash
haggletest -c -m 10 sub TEST
```

or alternatively

```bash
haggletest -u -q nop
```

to explicitly refresh the interest.

If node description refresh is enabled, 10 next objects are returned automatically with each refresh.

The following will clear the Bloom filter and return all local data objects (that have been previously fetched) plus any data objects from other nodes (in our case 10 from the publisher if executed on the other node).

```bash
haggletest -z -c nop
haggletest -m 10 sub TEST
```

Note that in general, since multiple nodes can contribute to answering a query, there is no guarantee that the number of data objects received by an application is bounded by the specified local bound. More answers can also be returned due to interest refresh mechanisms that are routing-algorithm dependent. Hence, -m 10 should be used to signal that the subscriber is interested in at least 10 results.
Furthermore, if we keep a standing subscription

`haggetest -z -c -m 10 sub TEST (no Ctrl-C)`

the next 10 results are returned and an attempt is made to route any results that become available to the subscriber without counting towards the bound. This can be observed by executing, e.g.,

`haggetest -b 5 pub TEST`

on the other node.

**Ranking of Results**

Results are ranked at each node first by matching-similarity (assuming the matching threshold is at least reached) and then by create time (representing the freshness of information).

Publish two data objects with X and Y attributes:

`haggetest -b 2 pub X Y`

Publish two data objects with X attributes:

`haggetest -b 2 pub X`

Query for data object with X and Y but only require 50% similarity:

`haggetest -z -c -t 50 sub X Y`

In our simple two-node network four results should be returned in the order:

1. last published data object with X Y (with SequenceNumber = 2)
2. first published data object with X Y (with SequenceNumber = 1)
3. last published data object with X (with SequenceNumber = 2)
4. first published data object with X (with SequenceNumber = 1)

However, since data objects are returned to applications in the order received, and multiple nodes can contribute to answering the query there is not guarantee that the ranking is always preserved.

**3.1.6 Basic Scenarios and Preliminary Results**

We constructed four simple scenarios to elucidate some basic properties of DIRECT. Note that these scenarios are not intended represent realistic military scenarios *per se*, although they may arise during a realistic scenario. They mainly serve only to compare against alternative routing protocols, and are to be used for micro-benchmarks and
correctness verification (including regression testing). These results are very preliminary and should not be taken too literally, they provide a useful exposé of the qualitative features of DIRECT, but further experimentation is necessary to make strong quantitative statements.

For all four scenarios, we used Linux containers in CORE and took the average of 3 runs. Within each run, we waited an initial warm-up period of several seconds until Haggle was fully initialized before publishing and subscribing.

In Static Scenario 1 there are 7 nodes, 3 publishers and 3 subscribers. Here, the shortest path between all of the publishers and subscribers is 2 hops through the middle-most node. In this case, Prophet will route all of the content through this middle node, whereas DIRECT can explore the alternative 3-hop path which may be less resource constrained. As the number of data objects published increases (despite the data object size), we expect the middle node to reduce Prophet’s performance faster than DIRECT’s, even though the three middle nodes belong to the same broadcast domain. We expect this reduction due to the increased CPU requirements for processing each data object.

In Static Scenario 2 there are 6 nodes, 1 publisher and 1 subscriber. As in the previous example, the purpose of this scenario is to demonstrate how DIRECT can utilize path diversity to reduce content delivery latency. Prophet uses the node between “P” and “S” to relay all of the data objects, where as DIRECT mainly uses this node as well, but also will explore paths through the 3 other nodes. Our preliminary results confirmed this hypothesis, and demonstrated that DIRECT had modestly lower average delivery latency than Prophet (0.996 seconds, versus 1.043 seconds, 100 1MB files, no fragmentation, 1 file published per second, ideal network).

Figure 5 Static Scenario 1: “P” represents publishers and “S” represents subscribers.

Figure 6 Static Scenario 2: “P” represents publishers and “S” represents subscribers.
In Mobile Scenario 1 there are 5 nodes, 1 publisher, 1 subscriber, and 1 mobile node. In this scenario, the mobile node travels between the publisher and the subscriber cyclically. The purpose of this scenario is to demonstrate how Prophet and DIRECT handle topology changes. Since Prophet uses a probabilistic, encounter-based method, it frequently attempts to deliver the content through the mobile node, leading to a high average delivery latency and in some cases failed delivery. In contrast, DIRECT avoids the mobile node and routes the content on the stable path between publisher and subscriber.

![Figure 7 Mobile Scenario 1](image)

**Figure 7 Mobile Scenario 1:** “P” represents publishers and “S” represents subscribers. There is a single mobile node that repeatedly moves between the publisher and the subscriber.

In Mobile Scenario 2 there are 4 nodes, 1 publisher and 1 subscriber. While the publisher is publishing a stream of content, the two middle nodes swap positions, temporarily interrupting the flow of content. Due to this topology change, Prophet must re-compute its probability matrix, whereas DIRECT utilizes the node description refresh to discover the new path. Despite a 30 second refresh period for DIRECT, preliminary results indicate that DIRECT can adapt modestly faster than Prophet in this scenario (4.310 seconds versus 4.798 seconds, 100 1MB data objects, no fragmentation, 1 data object published per second, ideal network).

![Figure 8 Mobile Scenario 2](image)

**Figure 8 Mobile Scenario 2:** “P” represents publishers and “S” represents subscribers. The two middle nodes swap positions once, disrupting the stream of content from the publisher to the subscriber.

### 3.1.7 Limitations and Possible Future Directions

Although our current version of content dissemination addresses several limitations of our previous versions, there are some remaining limitations that we summarize subsequently.

So far there have been no unforeseen problems with the proposed approach. Indeed, we have successfully integrated DIRECT relatively seamlessly with other complicated extensions, including network coding and fragmentation, demonstrating that the overall
design appears to have a good amount of modularity. However, there are some inherent disadvantages related to the underlying transport protocols. Unmodified Haggle uses host-to-host TCP connections to exchange node descriptions. This approach will not be effective in wireless scenarios where a single broadcast packet in DIRECT corresponds to a plurality of packets proportional to the number of neighbors. This increase in the number of transmissions will increase channel congestion and packet loss, increase delay, and consume more power than a broadcast approach to interest propagation (as described in DIRECT). To address this limitation of interest dissemination, we have implemented an initial prototype of UDP broadcast of node descriptions.

Future versions may want to keep track of more node description history details, instead of only maintaining the most recently received node description. As it stands, the current version does not support hybrid routing or multiple forwarding managers for the same classification. One could imagine extending this design to support a sequential application of forwarding managers and allowing multiple forwarding managers to be registered to the same classification. It might also be beneficial to extend the classifier module to use the event queue, to enable delayed forwarding and aggregation. In this case, the classifier would post forwarding events specifying the classification, while the forwarding modules would register to these events (as opposed to the classifier calling the forwarding module directly).

As future work, we will also consider exploring the pub-match and stop-match semantics for our version of DIRECT. We believe that our architecture is general enough that adding these features will not be difficult, and it would be useful to evaluate their potential efficiency advantages in the context of CBMEN.

**Proactive Replication** A key limitation to our approach of flooding as a means of proactive replication is its inherent wastefulness. Indeed, this approach cannot be used in scenarios with a large number of nodes, or where large data objects must be proactively replicated. We plan to address this limitation by enabling the user to specify the scope and frequency of the flood. For example, a user could proactively replicate a piece of content within only its immediate two-hop neighborhood. Similarly, we would like to explore hybrid routing that is dependent on the scope of the message: a data object could be proactively replicated within a small scope of the publisher, and routed using DIRECT outside of the scope of the publisher. Another interesting idea of a hybrid algorithm is reactive flooding, where the flooding is triggered only if some interest exists in the network. For future research, we may also explore more advanced forms of proactive replication that is integrated with the caching architecture. For instance, the decision whether to replicate content or not should be based on a notion of utility that more directly ties the algorithms to the needs of the systems and its users. Another interesting direction that might be of long-term interest is the use of distributed hash table techniques for storing content in specific regions of the network as a means of increasing scalability.
**Bloom Filters** Haggle uses Bloom filters (propagated as part of node descriptions) to avoid sending duplicate content. Specifically, before a node A sends a data object D towards another node B, A checks B’s Bloom filter for D in the node description in A’s data store. The implemented combination of DIRECT with Bloom filters is quite powerful, because unnecessary transmissions of potentially large data objects can be blocked even before they reach the neighbor of the requesting node. In fact this combination can be seen as an approximation of our interest-driven routing algorithm developed in the DARPA DTN program (Mark-Oliver Stehr, 2008), which used a separate knowledge dissemination algorithm to exchange cache summaries.

Nevertheless the overhead and the implications of exchanging Bloom filters need to be carefully examined. Currently, a Bloom filter fills several packets. Since each node description includes a Bloom filter, any efficient broadcast based dissemination of node descriptions should handle packet loss. Hence a naive replacement of TCP by UDP for node description exchanges may be insufficient if the loss rates are very high.

On the other hand, simply removing Bloom filters is insufficient to decrease the interest size to fit within a single packet. Haggle uses Bloom filters to prevent nodes from propagating node descriptions to a neighbor that has already received the description. If we remove the Bloom filters, we will have to add a mechanism to prevent loops and a broadcast storm of node descriptions.

We should also point out that there are some inherent limitations of Bloom filters, which can be a problem for some scenarios. Bloom filters can quickly check set membership, however there is a small probability of a false positive. Conceivably, albeit probabilistically unlikely, this situation could prevent a node from receiving important content, indefinitely.

Diving slightly deeper into the current implementation, Haggle nodes use Bloom filters that represent their neighbor's cache state to make forwarding decisions. This state can become invalid or stale if one is not careful to maintain its freshness. In certain situations the unmodified version of Haggle prematurely sets these Bloom filters, which can lead to invalid knowledge propagation and worsen the effects of false positives. To alleviate this problem, we have added a mechanism for periodically maintaining consistency with the counting bloom filter. We have also added configuration parameters to allow users to specify whether data objects should be removed from the Bloom filter when they are removed from the data store. For example, when the caching architecture purges stale data objects the user can now specify whether the purged data object remains in the Bloom filter, through the “keep_in_bloomfilter” option. Our long-term goal is to move towards a solution that will delete content from the Bloom filter because their capacity is limited. To avoid content circulation and strict time synchronization requirements this can only happen in a suitably delayed fashion.

Additionally, the further apart nodes are in the network, the more out of date the Bloom filters will become. Also, the more specific a query is (or the more unique a piece of content is) the less benefit there is to gain using Bloom filters. One can imagine queries
by an application that subscribes to a specific piece of content and then un-subscribes upon receiving that content: Bloom filters do not offer a clear advantage in this scenario. In this case, a few nodes will reply with the content and they will stop replying upon receiving the updated node description that indicates that the previously interested node is no longer interested in the content (at the same time they would receive the Bloom filter). Also, Bloom filters do not generally prevent the duplication that occurs when a node first adds an interest for data that it does not already have: without coordination, multiple nodes may still reply with the same content.

Interestingly, Haggle computes hashes over the content, the attributes, and the creation timestamp, when computing the content identifier, which also serves as a Bloom filter hash. Thus, if two data objects have the same content but different attributes then they will both be sent. From an architecture perspective this is necessary for correctness to prevent situations where nodes do not learn additional attributes for content due to a Bloom filter hit. This mechanism will not need to be changed to support the dissemination of content and metadata separately, because this can be done in separate data objects that are linked together, e.g., by a unique identifier.

In summary, we believe that further experimentation is necessary to evaluate the effectiveness of Bloom filters and the trade-offs of their use with respect to node description propagation and interest specificity.
3.2 Content-based Protocols: Lightweight Unicast and Broadcast

Unmodified Haggle supports several protocols for transporting data objects between a sender and receiver pair. The decision of which transport protocol to use is made after the data manager and the forwarding manager have determined what content to send, and where to send it. Unmodified Haggle has a simple policy that chooses the transport protocol based on the interface type. For example, unmodified Haggle will only use TCP communication between hosts that are connected through an Ethernet interface. The existing architecture employs a simple stop-and-wait protocol on top of the underlying network transport, to ensure data delivery and to reduce redundant transmissions. Note that both the stop-and-wait protocol and the network-layer transport protocol only occur between a sender and receiver pair when they are neighbors. Although this design may be effective in pocket-switched networks where contact times are long and infrequent, it must be adapted in order to efficiently support the unreliable broadcast scenarios that CBMEN targets. For example, the DIRECT routing algorithm uses broadcast messages to propagate interests, and will be inefficient when TCP is used as a substrate (especially in case the neighborhood is large). Additionally, the selection of which protocol to use should be based on the current state of the environment, and the content that is to be sent. To this aim, we have modified the protocol architecture to include unreliable UDP based protocols, and we have added a classification system that uses the content to be sent to dynamically select the transport protocol.

The stop-and-wait protocol works as follows: the sender initiates the communication by first sending the data object header. The receiver then uses the header to send either an ACCEPT or a REJECT to the sender, depending on whether the data object already exists at the receiver. If the receiver sends a REJECT message, then the sender assumes that the receiver already has the content, and marks the content as delivered. Otherwise, the sender proceeds to send the body of the data object. Upon successfully sending the body of the data object, the receiver will send an ACK message to notify the sender that the entire data object was successfully received. Note that, in the case of TCP network transport, these higher-level control messages operate in conjunction with the TCP control messages. After the sender receives the ACK, it may then send subsequent data objects using the same method. In the case of an error, there are protocol parameters that allow Haggle to retry this same process to send the data object (without keeping track of what has already been sent) after a specified delay. From an implementation perspective, communication only occurs between a sender and receiver instance, thus if two nodes, say A and B, are exchanging their node descriptions then there will be two separate TCP sessions between A and B. We have parameterized many of timers and retry counts used by this protocol, enabling users to specify them in the configuration file.
3.2.1 Light-weight Unicast and Broadcast via UDP

As a step towards UDP broadcast, we implemented a UDP unicast protocol that is an alternative to TCP. Note that unmodified Haggle has a UDP protocol, however it is insufficient for our purposes, since it is designed to only be used for communication between local Haggle applications and the Haggle daemon. Also, unmodified Haggle uses UDP for its neighbor discovery protocol, however it is not general enough or compatible with the existing protocol architecture. Our new protocol leverages the existing protocol architecture, and uses the same control protocol as described above, but it does not have the network protocol overhead that TCP incurs due to its congestion control and reliability requirements. With this implementation, when the protocol sends the header of the data object (or body), we pass the entire header (or body) to the UDP socket for transport. Thus, we rely upon the kernel’s use of UDP fragmentation for delivery. Note that this approach of relying upon implicit fragmentation may be ineffective in lossy channels, since an error, loss, or out of order delivery of any individual fragment will prevent delivery of the entire header (or body), although these limitations are less pronounced due to the hop-to-hop nature of the communication. We have been careful to design UDP unicast in way that minimizes the amount of code changes in the existing files. Our implementation of UDP unicast can be a useful alternative to TCP for small data objects and can be used in parallel with TCP with our new protocol architecture. However, the real advantage of UDP is its capability to naturally generalize to broadcasting which is a natural fit for content-based wireless networking and of particular interest for CBMEN.

As an initial version of UDP broadcast, we implemented UDP broadcast for node descriptions. This feature is paramount for data dissemination that requires flooding, such as DIRECT’s use of flooding interests. Our UDP broadcast implementation bypasses the higher-level stop-and-wait protocol, which is possible because unmodified Haggle treats an entire node description as data object header. After broadcasting a node description, we optimistically assume that the data object was successfully delivered to all of our known neighbors. To simplify the design, the protocol has a particular receiver in mind (the “reason” for the broadcast) which the sender believes does not already have the data object, but note this receiver is not indicated in the data that is actually transmitted over the wire.
3.2.2 Towards a Content-Based Protocol Architecture for Haggle

Figure 9 Refactored protocol design. For clarity, we have omitted some protocol classes that are irrelevant for this discussion.

Existing Classes

ManagerModule: All modules in Haggle descend from this class. It provides a thread that is associated with a specific Haggle manager, and a simple interface for configuration handling. Derived classes can retrieve the associated manager and post events to the main event queue.

Modified Classes

ProtocolManager: This manager is responsible for managing Protocol instances. It listens to interface events to instantiate server protocols that in turn are responsible for
instantiating clients. Sender clients are typically registered with the ProtocolManager, so that future communication with the receiver can use the existing instance, and avoid instantiating a new instance. We modified the protocol manager to use the ProtocolFactory to instantiate protocol instances, use the ProtocolClassifier to select protocols dynamically based on content, and instantiate the new UDP protocol proxies.

**Protocol**: This class is responsible for transporting data objects between a specific sender and receiver pair. It derives from ManagerModule, and runs in a separate thread. Additionally, protocol servers derive from this class. This class implements the stop-and-wait protocol. We have modified this class to use a configuration object for the stop-and-wait parameters, and retry parameters upon protocol failure.

**ConnectivityEthernet**: This class is responsible for discovering new neighbors over multiple Ethernet interfaces. Periodically, this class sends a UDP broadcast (known as a “hello” message) containing the sender’s MAC address, so that new neighbors can learn of its presence. We modified this class so that upon receiving a hello message we use the sender IP address and payload MAC address to perform an APR insertion, if the respective ARP entry is missing. This feature can be enabled or disabled in the configuration file.

**New Classes**

Note that the classifier classes parallel that of the forwarding architecture almost identically. As future work, we hope to unify these classes.

**ProtocolClassifier**: This abstract module is responsible for analyzing data objects and tagging them according to some criteria. Protocol modules can be registered with a specific tag, thus enabling dynamic content-based protocol selection.

**ProtocolClassifierAllMatch**: This concrete classifier module tags all content with a custom tag. The tag is specified in the configuration file. This classifier is useful for debugging and when used with the priority classifier.

**ProtocolClassifierAttribute**: This concrete classifier module tags content that has a specific attribute. Both the attribute and the tag are specified in the configuration file.

**ProtocolClassifierBasic**: This concrete classifier module tags content in two categories: “light-weight” and “heavy-weight”. Specifically, node description data objects are tagged as light-weight and all other data objects are tagged as heavy-weight. These tags are specified in the configuration file.

**ProtocolClassifierFactory**: This factory class is responsible for instantiating and configuring classifier objects given a specific classifier name.
ProtocolClassifierNodeDescription: This concrete classifier module tags node description data objects with a specified tag. This tag is specified in the configuration file.

ProtocolClassifierNodePriority: This meta-classifier allows the user to compose multiple classifiers according to a total order. Specifically, it resolves conflicts that may occur when two classifiers both have a valid tag for a single data object, by selecting the relevant classifier with the highest priority.

ProtocolClassifierSizeRange: This concrete classifier module tags content that is within a customizable size range. Both the range and the tag are specified in the configuration file.

ProtocolConfiguration: This concrete class is a container for protocol parameters. Protocol instances query the configuration for protocol related settings, allowing users to specify options such as retry counts and time out durations in the configuration file.

ProtocolConfigurationFactory: This factory class is responsible for instantiating configuration objects given a specific protocol name, based on the configuration file specification.

ProtocolConfigurationUDPBroadcast: This concrete configuration class is a container for UDP broadcast settings, such as the ports.

ProtocolConfigurationUDPUnicast: This concrete configuration class is a container for UDP unicast settings, such as the ports.

ProtocolFactory: This factory class is responsible for instantiating and configuring protocol objects given a specific protocol name. It registers these protocols with the classifier object, and passes a configuration object to the protocol.

ProtocolUDPBroadcast: This concrete protocol class implements an unreliable UDP broadcast protocol for node description transport.

ProtocolUDPBroadcastProxy: This concrete protocol class acts as a server, and acts as an interface for ProtocolUDPBroadcast to the ProtocolManager. It demultiplexes UDP packets to the respective ProtocolUDPBroadcast instance.

ProtocolUDPGeneric: An abstract protocol class that both ProtocolUDPUnicast and ProtocolUDPBroadcast derive from. It implements basic socket operations and error logic to work with the Protocol class.

ProtocolUDPUnicast: This concrete protocol class implements a unreliable UDP unicast protocol for arbitrary data object transport.
**ProtocolUDPUnicastProxy:** This concrete protocol class acts as a server, and acts as an interface for ProtocolUDPUnicast to the ProtocolManager. It demultiplexes UDP packets to the respective ProtocolUDPUnicast instance.

**SocketWrapper:** This is an abstract helper class to interact with Linux TCP and UDP sockets.

**SocketWrapperTCP:** A concrete wrapper class for TCP sockets.

**SocketWrapperUDP:** A concrete wrapper class for UDP sockets.

### 3.2.3 Sample Configuration and Parameters

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" />
Generic Parameters for 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. For instance, 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>
```
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:

```xml
<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, using e.g.,

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

Additional connectivity manager parameters in connection with UDP Protocols:

- `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 set to true.

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 receiver 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), i.e. pair of nodes, for UDP broadcast protocols with control (default 40). It is not recommended to increase this further 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.

```xml
<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.

### 3.2.4 Limitations and Possible Future Directions

**Limitations of the Initial Design** Perhaps the biggest limitation to our approach is the inherent unreliability with UDP. In the future, we may explore alternative control protocols to the stop-and-wait protocol that provide reliability and can exploit broadcast to gain additional performance. Also increasing reliability passively by overhearing the receiver’s retransmission might be worthwhile to explore. Our UDP broadcast implementation should also be extended to support broadcast of arbitrary data objects, and explicit fragmentation. This approach will increase network utilization by not discarding overheard information. For UDP unicast, our approach of passing the entire data object body to the UDP socket for implicit fragmentation limits the size of the data objects that can be sent with this protocol. These limitations make UDP practical for small data objects only.

Additional experimentation is required to evaluate the ramifications of broadcast interacting with the link-layer on the phones. In particular, broadcast may decrease battery utilization depending on its interaction with link-layer scheduling. To be sure, it will lead to an overall increase in the amount of packet processing by each phone.

Currently the decision of which protocol to use is only determined by examining the data object that is to be sent. A useful extension would be to expand our classifiers to also take into account the environment, history and resources available between the communicating nodes. Note that the classifier architecture parallels that of the forwarder classifier architecture. In the future, we hope to merge these into a single set of classifiers.
When implementing the proposed design, several unexpected problems arose with UDP and link-layer interactions, which led to poor performance. Specifically, our initial implementation had poor performance if the ARP cache entries between the communicating parties were nodes. To overcome this limitation, we exploit the fact that the Haggle architecture only uses single-hop host-to-host communication, thus the Haggle application has the necessary (IP, MAC) mappings before the ARP cache is populated. We have added a mechanism for Haggle to exploit this information by manually inserting these entries into the ARP cache. For security reasons, we have implemented this insertion process as a separate C program, that is setuid to root. Note that this problem does not arise in the case of TCP-only communication, since TCP has reliability guarantees and the kernel will buffer the TCP packets until the ARP cache is filled.

The rapid increase of packet losses due to UDP fragmentation is a well-known limitation of both UDP unicast and broadcast. Note that the use of TCP will avoid fragmentation by only sending packets smaller than the MTU and hence will be advantageous in many cases up to a certain number of neighbors.

Another limitation of UDP broadcast in 802.11 wireless networks is the lack of RTS/CTS signaling and link-layer acknowledgements, which makes it inherently less reliable than UDP unicast. Interestingly, the corresponding losses increase with medium contention and hence with the number of neighbors. Our flexible current protocol architecture allows us to avoid an a priori commitment to a particular protocol and can be a stepping stone for a future design where the decision which protocol to use can be made at runtime on a range of factors and optimization objectives. Another practical approach to improve UDP broadcast is to use promiscuous mode, which could be naturally implemented as a hybrid between our UDP broadcast and unicast protocols. The other promising direction is to insert an additional layer above UDP like NORM, which provides better reliability though forward error correction and optional (negative) acknowledgements.
3.3 Content-Based and Utility-Based Caching

Since content-based mobile edge networks can persistently store large amounts of content, a key concern is to develop mechanisms that ensure that during the operation the amount of content managed by the network does not grow beyond its bounded capacity and resources are primarily used for content that is relevant to the user. Unlike IP-based network, content-based networks maintain explicit information about the meaning and the use of content at a level of abstraction corresponding to semantically meaningful units that can be exploited for better content-management and as a step towards semantic networking.

The goal of content-based caching is to design efficient and effective content-aware caching algorithms inside the Haggle framework. The content-based caching algorithms we are developing will initially be opportunistic (or reactive), meaning that they will make caching-related decisions for content that is already located or flowing through the node on which they are running. The same algorithms can be applied together with proactive dissemination of content, but since more content is flowing through the network it becomes important to make fine-grained decisions of what content to cache and quickly discard content more actively that does not have sufficient utility. The decision whether such an action is economical needs to be based on the consideration of benefits and costs for each individual piece of content in a context-dependent and cooperative manner. In this preliminary design, we will focus on the opportunistic class of algorithms that has already been implemented.

3.3.1 Content-Based Caching Approach

Unmodified Haggle provides a mechanism to specify priority of an interest using “weights”. This mechanism is useful for prioritizing which data objects are sent first from a sender to a receiver, or limiting the number of data objects that are sent. Beyond the weights, Haggle does not natively support a mechanism for a user to specify the relevance of a class of content, or a way to specify the relative “freshness” of two pieces of content in the same class.

On the other hand, caching the right information is of key importance to improve latency in view of limited networking bandwidth and potentially costly content query operations. Hence, a more fine-grained mechanism is needed to give users control over how long and which data objects are cached at a particular node. The benefits of such a mechanism includes: minimizing network load (irrelevant data will be discarded sooner), device storage usage, and latency (more bandwidth will be available, and fewer data objects will be in the data store). Caching techniques and their trade-offs have been studied extensively in research such as (K. Obraczka T. Spyropoulos, 2009). The ideal design for CBMEN should be generic enough to support different caching algorithms from this body of research, and should fit naturally within the Haggle framework.

Content-Based Caching Strategy To solve the limitations of unmodified Haggle, we have 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. The architecture is extensible so that one could easily add a module that treats Blue-Force tracking data differently than FTP or web content. Specifically, Blue-Force tracking data may have requirements such as low latency, low jitter, and faster aging (retransmissions are not necessarily useful), while FTP or web transfers do not necessarily have jitter or latency requirements.

Administrators will use the configuration file to specify the cache strategy module, and its respective submodule. 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 component and the cache purger component.

**Content-Based Replacement** If the replacement component is selected for handling the content, the cache strategy module will pass the newly received data object to it. The replacement module is responsible for deciding whether to insert or discard the newly received data object. If the module inserts the data object then it is also responsible for deleting existing stale content from the data store.

While the target architecture is open-ended to support multiple types of replacement policies, our first implementation uses a total order replacement algorithm. Total order replacement will keep only the “freshest” piece of content, while “staler” content is either dropped or deleted from the data store. The configuration file is used to specify a tag name which denotes that a tagged data object is totally ordered, an id attribute which indicates the content originator, and a metric field name which the module uses to determine the freshness of the object.

In addition to the total order replacement component, we have added a priority replacement module that can be composed of multiple total order replacement modules. This functionality allows customizing the configuration file to specify complex replacement rules, such as lexicographical ordering. We explain the use-case of composing multiple replacement modules through priority policies in a separate subsection below. Another potential dimension for generalization that might be of future interest is the ordering itself. More general orderings are conceivable, including the most general case of a user-defined partial order as in our partially ordered knowledge-sharing model that has been developed in (Minyoung Kim, 2010).

**Content-Based Purging** Content can be purged either by absolute or relative expiration times. An absolute expiration time is specified as a predefined time (e.g., Jan 30, 2013 at 5:15pm). A relative expiration time is specified as a minimum time to live (in seconds) from the reception, which allows data objects reasonably sufficient time to propagate yet aims to control the amount of cached content, e.g., to reduce access times for incoming data objects. Purging enables Haggle to expire data (when no longer needed) and to provide better use of resources. Examples of this are stale orders (e.g., inactive missions) and obsolete information (e.g., old map, location, communication codes).
Content-Based Priority Policies In our current implementation, the cache replacement modules can be prioritized when composed into a complex cache strategy. When a data object is examined to determine which replacement module to apply, priority determines how to proceed. For example, assume there are three different replacement modules R1, R2, R3, where R1 is the highest priority and R3 is the lowest priority, that are applicable to a data object with an attribute (A). The replacement module R1 with the highest priority will be checked first, to see if it is responsible for the data object. If the attribute (defined in the configuration file) is matched, R1 will handle the object accordingly (e.g., total ordering on the attribute A). Otherwise, it will proceed with the next highest priority (R2). Our sample configuration at the end of this section specifies replacement mechanism with priority policies based on mission timestamp and content create time.

Content-Based Parallel Policies Logically speaking, the cache replacement modules operate sequentially to make caching decisions (because they may depend on each other), while the cache purger modules operates in parallel (purgers are considered to be independent of each other). Thus, with multiple purger policies, the data object can be acted upon by each of the policies. Parallel policies purge when at least one of parallel policies dictates to purge. Purging can be triggered for many conceivable reasons such as creation time, expiration time, content type, or locality of content (e.g., out of scope relative to a mission or a named geographic area). Since expiration time and locality are independent measures, purging conditions can be checked in parallel for the same data object. Our sample configuration at the end of this section illustrates the specification of a purging mechanism with parallel policies based on absolute and relative expiration times.

Content-Based Lexicographical Replacement As first step toward generalization of the ordering, we implemented lexicographical rules by using the aforementioned prioritized replacement policies. When a data object is received, it uses the priority to decide which algorithm may act upon the data object. If multiple cache replacement orderings are applicable to a data object, the priority policies can be used to define a lexicographical ordering. The replacement order with the highest priority first examines the first element to be ordered. Subsequently, replacement ordering with the n-th highest priority examines the n-th element.

Let us use an example with two attributes, in (X, Y) format, where the first attribute X is handled by replacement order O1, and Y is handled by replacement order O2 in case X cannot resolve the ordering between two data objects: Suppose a data object with attributes of (A, B) is already cached in the Haggle. Now, a new object is inserted to the system with attributes (C, D). The following lexicographical rules are applied:

First, we compare A to C by applying O1.
   a. If A > C, drop the object with (C, D)
   b. Else, if A < C, delete the object with (A, B), insert the object with (C, D)
   c. Otherwise, we compare B to D by applying O2.
      i. If B > D, we drop the object with (C, D)
ii. Else, if B < D, we delete the object with (A, B), insert the object with (C, D)
iii. Otherwise, we insert the object with (C, D).

In case only one replacement ordering can be applied, i.e., a data object with a single corresponding attribute, only the corresponding ordering is applied. Suppose a data object with attributes of (A) is already cached, and a new object is inserted to the system with an attribute (C).

a. If A > C, drop the object with (C).
b. Else, if A < C, delete the object with (A), insert the object with (C).
c. Otherwise, we insert the object with (C).

Note that the rules for multiple cache replacement orderings are applied per exact matches. Suppose Haggle already caches a data object with (A1, B), and a new object is inserted with (A2). In this case, both data objects will be kept in the cache (regardless of the comparison between A1 and A2) since the ordering cannot be applied. For the same reason, the orderings O1, O2, O3 that apply to data objects with attributes (X, Y, Z) respectively, will not be applied to data objects with different set of attributes. A data object that is applicable to O1 and O3 will only be compared to other data objects that have the attributes (X, Z) without having (Y).

We give a mathematical formulation for the composition of total order replacement and priority replacement at the end of this section.

3.3.2 Utility-Based Caching Framework

The goal of utility-based caching is to find the optimal set of data objects that fit in the fixed-sized data store. The main idea is to periodically compute a utility (a real between 0 and 1, where 1 has highest utility) and evict data objects from the cache that do not meet a sufficient utility threshold, or evict data objects with the least utility in the case where the capacity is exceeded. This mechanism allows Haggle to more intelligently manage disk space resources. The design extends the content caching design, and we implemented utility-based caching as a cache strategy in the existing design. Previous strategies (e.g., total-order replacement and time-based purging) now become special cases of utility-based caching. The strength of our method lies in combining multiple sources of knowledge to influence caching decisions, and the ease and flexibility in specifying the utility function (declarative approach).

The basic pipeline is as follows:
1. Compute Utility: computes a utility for each data object in the cache using a specified utility function. Each data object is assigned a double in the range \([0,1]\).

2. Threshold Filter: all data objects with computed utility less than this threshold are immediately evicted, regardless of the cache capacity.

3. Plan Generator: in the case where the water-mark cache capacity is exceeded, we frame the problem of selecting which data objects to keep in the cache as a 0-1 knapsack problem with the utility as the benefit, and the size of the data object as the cost. This stage uses a (typically heuristic-based) knapsack optimizer to calculate the set of data objects to evict.

Currently, we support the described caching pipeline with the utility functions: popularity (LRU-K or LRFU), and neighborhood (milestone 1). We have implemented a fixed weight global optimizer, whose settings are configurable via config.xml. We implemented the heuristic-based knapsack optimizer, which greedily fills the knapsack using highest marginal utility (utility/cost) first (until the watermark capacity is surpassed).

The pipeline can be executed either periodically, or whenever a data object is inserted (event based, which occurs upon receiving a data object from a remote node), or both. The watermark capacity functions as a soft constraint, it may be exceeded, but it will not be exceeded after an execution of the pipeline. The maximum capacity specification is a hard constraint, data objects will be dropped without execution of the pipeline if this constraint is surpassed. A typical use case is for a data object to be inserted which causes a watermark constraint violation but not a hard constraint violation, then upon pipeline execution data objects will be evicted so that the watermark constraint is no longer violated.

To avoid frequently computing redundant utility values, a data object’s utility has a specified age \(X\) (specified using “compute_period_ms” in config.xml) which will prevent a re-computation of the utility if the utility was computed within the past \(X\) milliseconds.
The default greedy knapsack optimizer will break ties using the creation time. Note that an empty aggregate utility function and the greedy optimizer will treat the cache as a FIFO queue based on data object creation time (this is similar to the default Haggle behavior, but with an explicit cache resource constraint bound).

All utility functions return a value between [0,1], while some utility functions return only 0 or 1. A value of 0 is interpreted as the utility function designating that the data object has no utility (it should not be stored in the cache). While a value of 1 indicates that the data object has a lot of utility and should be kept in the cache.

We support the following utility functions:

- **CacheUtilityAggregateMin**: takes the minimum utility value of its constituent utility functions.
- **CacheUtilityAggregateMax**: takes the maximum utility value of its constituent utility functions.
- **CacheUtilityAggregateSum**: sums the values of its constituent utility functions, and ensures the compute utility is within [0,1].
- **CacheUtilityNeighborhood**: computes a utility based on the frequency of the data object within the 1-hop neighborhood (by inspecting the Bloom filter) and hence specifies a form of cooperative caching. This is usually used to reduce the utility of keeping a particular data object that frequently appears within a neighborhood.
- **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 to a peer.
- **CacheUtilityNewTimeImmunity**: gives a positive value if the data object was received within a certain timeframe. Default is 2.5 seconds, but can be adjusted with xml option. Thus, all data objects 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 on full systems, large new data objects were dropped before they had a chance to disperse, as older data objects were distributed several times, giving them higher values.
- **CacheUtilityPurgerRelTTL**: computes a {1,0} utility based on whether the data object has expired by a relative received time. 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".
- **CacheUtilityPurgerAbsTTL**: This utility function is identical to the RelTTL, but uses absolute time.
CacheUtilityReplacementTotalOrder: computes a \{1,0\} utility based on whether the data object 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".

CacheUtilityReplacementPriority: 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.

CacheUtilityAttribute: computes a utility based on a specific attribute. If Haggle receives a data object 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 to divide the utility to compute the utility over an attribute that isn't necessarily normalized to [0,1]. Default is 1.

### 3.3.3 Applications of Content-Based Caching

Total-order-based replacement is useful for applications such as GPS location tracking, situation-awareness/surveillance (e.g., latest data from a stream of time-stamped images of a location or object), and real-time measurements (e.g., latest temperature of an area or device). For example, suppose there is a Blue-Force tracking application running on phones that periodically sends out a location beacon containing GPS information to every other phone in the network. In this scenario, the Blue-Force tracking application only cares about the most recent GPS information, and it uses total order replacement to specify this preference to the network. The total order replacement module will then discard beacons for a particular phone that is less recent than an already received beacon for the same phone. Similarly, upon receiving a more recent beacon for a specific phone than what has previously been received, the module will insert the new beacon data object into the data store and remove the old beacon data object. In this way, old beacons are removed at the earliest possible time from the network (rather than by applications at the end points) so that they consume fewer resources, while new beacons are given priority and have more resources to propagate.

As an extension to the Blue-Force tracking based on total order replacement, expiration times can be used for purging potentially sensitive content. In a situation when any soldier loses his/her smart device that contains all traces of Blue-Force locations and movements in the field, purging based on expiration times can prevent information from being revealed to a certain degree, even if the security was compromised. Examples can include operation orders, pictures of sympathizers, location of mine fields, and other confidential/secret information. This potentially sensitive information could be used against friendly forces and the impact can be devastating. Under those circumstances, enforcing content purging based on expiration time that is specific to the type of content can help to limit the leakage of sensitive content independent of other cryptographic security mechanisms.
Furthermore, by combining replacement with purging, a mechanism similar to a watchdog timer can be enforced. Pre-existing orders with an expiration time can be extended, in case the mission is active longer than expected. Protecting communication codes (which in many cases are associated with expiration times) can be another use-case. In a situation when a communication code is broken, one needs to resend a newer code that will replace the old code regardless of its expiration time.

Lexicographical ordering in our cache replacement framework can be used to implement fragmentary order updates. For example, the update priority can be the first attribute to be compared and the update time can be the second attribute in the lexicographical ordering. If the commander publishes content for fragmentary order updates with \textless priority, time\textgreater attributes, then the higher priority updates will replace updates associated with lower priorities. For updates with same the priority, the update time determines which content is replaced (i.e., more recent updates replace older ones, within the class of orders that have the same priority). Other possibilities include security call-signs and last minute intelligence updates (e.g., a mine field location update).

3.3.4 Towards a General Caching Strategy Architecture for Haggle

In the same way that the forwarding manager allows a developer to specify a forwarding module (i.e., ForwarderProphet or ForwarderAlphaDirect) in the configuration file, we added a cache strategy module to the data manager.

As with the forwarding modules, this module is asynchronous, and runs in its own thread so that the additional processing does not block the data manager or any other managers that a running in the Haggle kernel. The cache strategy module has its own event queue to serialize database accesses, which may not otherwise be atomic. For example, in the case where a new data object is received which subsumes an existing data object, the module must be careful to delete the stale data object and insert the new data object, in a single atomic operation; otherwise, there may be consistency errors such as the existence of a stale data object in the data store, or duplicates of the same data object.

The data manager has been augmented to pass a newly received and verified data object to the cache strategy module. In the current implementation, the concrete cache strategy is implemented by the CacheStrategyReplacementPurger, which runs in its own thread and passes the data object to the cache purger and cache replacement modules.

To support the total order replacement module, we augmented the data store and SQL data store to provide a helper function for this replacement module that returns all of the data objects that the replacement module is responsible for, sorted by freshness. This functionality has been implemented by adding an additional SQL statement to the SQL data store (specifically the SQLDataStore module). The total order replacement module is responsible for deciding whether or not to insert a newly received data object, and if it
does insert it must decide what data objects to remove. Additional query and query result classes have been added to the data store (specifically the DataStore module) to allow the replacement modules to communicate with the data store and the data manager across callbacks.

Independent of the replacement, we have added cache purger modules to provide an even finer control of caching based on expiration time. In this case, data objects are evicted from the data store after they have surpassed a specified age. Unmodified Haggle does not purge content, except after reaching a global maximum age. The new type of functionality requires the purger module to support timer-based events for already cached data objects, as opposed to the replacement modules, which only need to handle data objects when they are received.

In utility-based caching, a special case that might be of interest is content that has missed an optional (soft) delivery deadline (where the deadline can be content-dependent) that does not have much remaining utility and could be purged in the near future to free storage and bandwidth for other traffic.

![Figure 10 Class Diagram for the Cache Architecture](image)

**Figure 10** Class Diagram for the Cache Architecture

**Existing Classes**

**ManagerModule:** All modules in Haggle descend from this class. It provides a thread that is associated with a specific Haggle manager, and a simple interface for
configuration handling. Derived classes can retrieve the associated manager and post events to the main event queue.

**New Classes**

**CacheStrategy**: All cache strategy modules in Haggle are descended from this abstract class. It provides a common interface for cache strategy modules, so it can identify and process them best. Actual cache strategy occurs within the identified correct cache strategy module.

**CacheStrategyAsynchronous**: This abstract cache strategy module serves as a base class for other cache strategy modules that need asynchronous functionality. This class runs in its own thread, running asynchronously to the Haggle kernel. However, it will process data objects in order (synchronous for cache strategy modules).

**CacheStrategyReplacementPurger**: This cache strategy contains a single cache purger and a single cache replacement. Note that the cache purger can be composed of multiple cache purgers, through the parallel purger. Similarly, the cache replacement can be composed of multiple cache replacements, through the priority replacement.

**CacheReplacementPriority**: This module handles the priority of cache replacement modules. When a data object is received, it goes down the cache replacement priority list. The replacement module with higher priority handles the data object. A lexicographical ordering can be specified via the priorities in the configuration.

**CacheReplacement**: All cache replacement modules in Haggle descend from this abstract class. It provides an interface for the replacement module, so that it can identify the type of data objects received and decide how to process them. Specifically, it will defer processing decisions to a replacement module whenever an exact attribute matching (for the specified module) occurs. The actual processing occurs after a data object is identified, via attributes, as being a special case, and needing to be handled by the specified module. More generally, our design also allows for a set of concrete module(s) to act upon the data object.

**CachePurgerParallel**: This module determines which purger module will be used upon a data object. Unlike the CachePriority modules, which will process based upon a configuration order, the parallel module will process all relevant data objects.

**CachePurger**: All cache purger modules in Haggle descend from this abstract class. It provides an interface for the purger module, so that it can identify the type of data objects received and how to best process them. Specifically, it will defer processing decisions to a purger module whenever an exact attribute matching (for the specified module) occurs. The actual processing occurs after a data object is identified, via attributes, as being a special case, which is handled by the specified module. More
generally, our design also allows for a set of concrete module(s) to act upon the data object.

**CacheReplacementTotalOrder**: This concrete replacement module identifies specifically marked data objects, and compares the data object to the data objects in the data store, based upon the originator. When applied to time stamps, only the “freshest” content is preserved, and the “staler” content is discarded. Thus, it is useful when you only need the latest information, such as GPS location or status reports.

**CachePurgerAbsTTL**: This concrete purger module identifies specifically marked data objects. The data object will be purged based on absolute expiration time. It is useful to expire information that is no longer necessary, as opposed to letting the system guess which data is not necessary.

**CachePurgerRelTTL**: This concrete purger module identifies specifically marked data objects. The data object will be purged based on relative expiration time based upon its reception. It is useful to give a minimum time to live for short-lived content (e.g., status updates), after receipt of the data object, and to control dissemination of the data object.

**CacheStrategyFactory**: This factory class is responsible for instantiating and configuring cache strategy objects at initialization time.

**CacheReplacementFactory**: This factory class is responsible for instantiating and configuring cache replacement objects at initialization time.

**CachePurgerFactory**: This factory class is responsible for instantiating and configuring cache purger objects at initialization time.

**CacheStrategyUtility**: This class implements the utility-based content pipeline. It derives from the CacheStrategyAsynchronous class (derived from the CacheStrategy abstract class) and allows a user to specify a maximum cache capacity and a watermark to bound the amount of space used by data objects with files. The pipeline may can be executed whenever a new data object is inserted, or periodically (or both). Upon pipeline execution (or data object insertion), a user specified threshold allows immediate eviction from the cache. All data objects who have not recently had their utilities computed will then have their utilities recomputed using the specified utility function (which may be an aggregate of other utility functions, and may use the global optimizer for weights), and if the watermark is exceeded then the knapsack optimizer is activated to select a subset of the data objects to be evicted (least utility first) until the watermark is not exceeded.

**CacheUtilityFunction**: This class provides an abstract utility function class. A utility function simply returns a value between 0-1 for a particular data object, and uses a global optimizer to weight this value. A lower value means that the data object does not
have very much utility for storage at that node, and it should be evicted from the cache before a data object with higher utility. This file implements the utility functions:

1. CacheUtilityAggregateMin/Max/Sum,
2. CacheUtilityNeighborhood,
3. CacheUtilityPopularity,
4. CacheUtilityNewTimeliness,
5. CacheUtilityPurgerRelTTL/AbsTTL,
6. CacheUtilityReplacementTotalOrder,
7. CacheUtilityAttribute,
8. CacheUtilityRandom, and
9. CacheUtilityNOP.

(1) are basic combiners for other utility functions, and return the min/max/sum of its constituents and ensure the value is in [0,1]. (2) uses Bloom filter neighborhood information to typically decrease the utility of a data object. (3) uses data object usage metrics to perform either LRU-k or LRFU. (4) gives a positive value if the DO was received within a certain timeframe. (5) computes a 1,0 utility based on whether the data object has expired by a relative/absolute received date. (6) computes a 1,0 utility based on whether the data object is subsumed by another data object. (7) computes a utility based on a specific attribute. (8) computes a random utility in [0,1]. (9) corresponds to the NOP function (returns 1).

**CacheUtilityFunctionFactory**: This factory class instantiates and configures utility function classes by name.

**CacheGlobalOptimizer**: This class implements CacheGlobalOptimizerFixedWeights concrete class that allows a user to specify fixed weights (in range [-1, 1]) for utility functions, via config.xml.

**CacheGlobalOptimizerFactory**: This factory class instantiates and configures global optimizer classes by name.

**CacheKnapsackOptimizer**: This class provides an abstract class for knapsack optimizers. Practically speaking, this class returns a list of data objects to be removed from the cache. This file implements the "CacheKnapsackOptimizerGreedy" to heuristically fill the cache by inserting data objects with the highest marginal utility first (utility/size), until the capacity constraint is violated. Upon ties, data objects that were created earlier are inserted first (creation time).

**CacheKnapsackOptimizerFactory**: This factory class instantiates knapsack optimizer classes by name.

**EvictStrategyLRFU**: This module implements LRFU function, calculating a raw score. LRFU is based, roughly, on the formula
CRF(now) = f(0)+f(now-lastaccess)*CRF(lastaccess), where f(x) = (1/p)^<=(lambda*x).

Note: CRF = combined recency and frequency.
The value ONLY changes after an access.

EvictStrategyLRU_K: This module implements the LRU-k. In simple terms, LRU-k keeps track of past k accesses. Evictions are based on the k-th-to-last access timestamp.

EvictStrategy: This module is a base class for the popularity-based cache eviction modules, in particular, to both EvictStrategyLRFU and EvictStrategyLRU_K modules.

EvictStrategyManager: This module manages the various popularity-based cache eviction modules. When a data object is matched with a node, it is sent to this module. This module will pass the data object to each LRU module to update the internal scratchpad information. In addition, this module can be set to a default popularity-based cache eviction, so only the default LRU module results will be returned per data object.

EvictStrategyFactory: This factory class instantiates LRUEvictStrategyManager, and populates it, based upon config.xml, with LRFU/LRU_K EvictStrategy modules.

Modified Classes

DataStore and SQLDataStore: The API has been extended to support a new query method doDataObjectQueryForReplacementTotalOrder together with a class representing results of such queries and a corresponding task for asynchronous processing.

3.3.5 Sample Configuration and Parameters

The caching strategy is configured by the <CacheStrategy> tag in the configuration file. An excerpt is shown below: (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.

```
<CacheStrategy name="CacheStrategyReplacementPurger">
  <CacheStrategyReplacementPurger purger="CachePurgerParallel" replacement="CacheReplacementPriority">
    <CachePurgerParallel>
      <CachePurger name="CachePurgerAbsTTL">
```

Distribution Statement A: Approved for Public Release, Distribution Unlimited

67
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:

\[
\text{utility}(d) = \min( \\
\text{\quad \quad max}(0.3*\text{nbrhood}(d) + 0.10*\text{random}(d) + 0.70*\text{pop}(d) + 0*\text{NOP}, \text{newtimeimmune}(d)), \\
\text{\quad \quad min}(\text{relttl}(d), \text{absttl}(d)), \\
\text{\quad \quad 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.

```xml
<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"/>
  </CacheStrategyUtility>
</CacheStrategy>
```
<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_probabilistic="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>
This above configuration file also contains new parameters defined as follows:

<CacheStrategy>

name - we implemented utility based caching 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, then the default name 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": enable/disable 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": enable/disable 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 system debugging only.

manage_only_remote_files - "true" or "false": enable/disable management of locally published files in addition to files received remotely. In most cases this value should be "true". "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 - "true" or "false": if "true", then data objects evicted will remain in the bloomfilter, otherwise they are removed.

bloomfilter_remove_delay_ms - must have keep_in_bloomfilter="false".
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* - when the knapsack optimizer computes the marginal utility, it will use this parameter to discretize the sizes of the data objects to a less granular level. For example, if the discrete_size is set to 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).

### 3.3.6 Limitations and Possible Future Directions

The currently implemented replacement module, total order replacement, only keeps the “freshest” data object for a particular class of content. It may be beneficial to generalize this module further to support a window of content, such as the n freshest data objects. In GPS tracking applications, such functionality may be useful to calculate estimates such as the speed and direction of a tracked device. Other conceivable replacement modules might include replacement by hop-count or GPS location, and the ability to specify a maximum number of data object replicas in the network. Ideally, these replacement modules should also incorporate a notion of content-based utility in order to quantify the utility of storing or evicting a particular data object.

The current design is for opportunistic caching, but together with the utility-based optimization approach it naturally generalizes to proactive and cooperative caching. In addition to traditional measures of utility, one future source of utility can be the interest model discussed in the next section. Proactive caching can support proactive replication of content based on suitable utility functions, which take into account benefit and cost. The later is important in proactive replication due to potentially high bandwidth consumption. Cooperative caching can be seen as a special case of utility-based caching and can furthermore take into account the spatial distribution of content that is already cached elsewhere, e.g. by inspecting the Bloom filters of other nodes.

### 3.3.7 Formalization of Total-Order Replacement

To avoid any ambiguity that may arise in the informal description, we subsequently give a mathematical formulation for the composition of total order replacement and priority replacement.

Let $D$ be the set of data objects $D = \{d_1, \ldots, d_n\}$.

Let $a_j$ be an “attribute” function $a_j : D \rightarrow \mathbb{N}$. We interpret $a_j$ as mapping from a metric attribute to its value.

Let $f$ be an “attributes” function $f : D \rightarrow 2^A$, where $2^A$ represents the power set of $A$. We give $f$ the interpretation of mapping a data object to the set of “relevant” attributes (these
are the attributes that are specified in the data object and are eligible for total order replacement).

We say that two data objects, say $d_j, d_k$, belong to the same “type” equivalence class, denoted $d_j \equiv_t d_k$ if and only if they have the same attribute set:

$$\forall d_j \in D \forall d_k \in D [d_j \equiv_t d_k \iff f(d_j) = f(d_k)]. \quad (1)$$

Note that $\equiv_t$ forms an equivalence class on $D$.

Let $T$ be a totally ordered sequence of $A$, in other words $T = (a_1, a_2, \ldots, a_m)$. We say that $a_i <_T a_j$ if and only if $i < j$ in sequence $T$. Informally, we say that attribute $a_i$ has higher priority than attribute $a_j$.

Let $2^T$ be the set of all ordered subsequences of $T$. Let $S_T: 2^A \rightarrow 2^T$ be the mapping of the unordered set of attributes to its ordered equivalent under $T$.

Let $O_T: D \rightarrow 2^T$ be a function that maps a data object to its totally ordered set of attributes:

$$\forall d_j \in D [O_T(d_j) = S_T(f(d_j))]. \quad (2)$$

Given two ordered sequences of integers say $P, Q$ with equivalent length $|P| = |Q| = z$. We assign a total ordering $<_S$ as follows: $P = (p_1, p_2, \ldots, p_z), Q = (q_1, q_2, \ldots, q_z)$.

$$P <_S Q \iff \bigvee_{i \in \{1,\ldots,z\}} \left[ \left( \bigwedge_{j \in \{1,\ldots,i-1\}} p_j = q_j \right) \land p_i < q_i \right]. \quad (3)$$

In other words, $<_S$ is a lexicographical ordering on sets $P$ and $Q$. $<_S$ is undefined if $P$ and $Q$ do not have the same number of elements.

Let $C$ be the function that maps a data object to an ordered sequence of natural numbers, according to $O_T$. In other words,

$$\forall d_j \in D [C(d_j) = (n_1, n_2, \ldots, n_m) \iff (O_T(d_j) = (a_1, a_2, \ldots, a_m)$$

$$\land [\forall i \in \{1,\ldots,m\} (n_i = a_i(d_j))]]. \quad (4)$$

Finally, we order data objects, denoted $<_T$, as follows:
∀d_j ∈ D∀d_k ∈ D[(d_j < d_k) ↔ (d_j ≡_t d_k ∧ C(d_j) <_s C(d_k))]

The total order replacement policy simply keeps the greatest data object (under < _T ) within each equivalence class. Formally, total order replacement on a non-empty data store D maintains the constraint:

\[ |D| > 0 \land \forall d_i \in D \forall d_j \in D(d_i \equiv_t d_j \rightarrow d_i = d_j) \]  

Let D be the set of all possible data objects, and let D^i ⊆ D be a non-empty data store at time i, and let d^i be the data object which is received at time i.

\[ d^i \in D^{i+1} \iff [\forall d_j \in D^i (d_j \equiv d^i \rightarrow [d_j = d^i \lor d_j <_t d^i])]. \]

Equations 6 and 7 are the constraints that total order replacement imposes on the data store.
3.4 Interest Management and Adaptive Interest Modeling (AIM)

This section describes the design of the interest management component which is an important part of the CBMEN ENCODERS solution and includes SAIC’s Adaptive Interest Modeling (AIM) approach. Different from earlier applications of AIM at SET Corp. to analyze traffic of chat applications, SRI and SET have developed in joint work a novel way to leverage AIM in the context of content-based publish/subscribe systems such as Haggle. The key novel idea is to classify the current user model based on a set of pre-defined or pre-trained situation-dependent profiles. The results of that classification can then be used to actively trigger other actions such as smart prefetching of relevant content. The latest version of our interest manager also supports assignment and subscription to missions, which are profiles that can be activated independently of the interest model.

3.4.1 AIM Approach

A key premise of ENCODERS is that accurate, real-time understanding of edge users’ dynamic interests and information needs can be leveraged:

- To provide efficient and effective content management by distributing only relevant content rather than flooding the network, i.e. interest model capabilities can be leveraged to make caching and content distribution decisions tailored to the warfighter’s information needs. They can also be utilized by network services for smarter routing decisions.
- To enhance security by detecting potentially anomalous queries (Note: this is a Phase 2 application and will not be described herein)

Dynamic, adaptive modeling of a users’ information context is key to this understanding and that is precisely what AIM is intended to provide.


AIM adapts to the MANET environment by updating interests and their weights based on node and network events (e.g., publish and subscribe events) that reflect node’s interests and information needs. More specifically, the ARAMA algorithm will process these events to adapt the interest model for the local node, which in turn will be used to construct its VIG. Most importantly for CBMEN, the interest model and the VIG can be used to influence opportunistic/proactive caching and prefetching decisions.

AIM applies these adaptive user modeling algorithms to dynamically create an Interest Model (IM). AIM can be applied at multiple levels:

- individual node (an individual warfighter),
- a cluster of nodes (unit), or
- cross-cluster group with similar interests (VIG - virtual interest group).
Interests in the model indicate the warfighter’s (or cluster’s or group’s) information needs in terms of topics, named entities (people, places, organizations, etc.), and more importantly, the relationships among them, which represent interest correlation. Each interest element is associated with a dynamic weight indicating the level of relevance.

Note that an individual node is normally associated with a particular warfighter. Further that a cluster of nodes is associated with a warfighter unit such as a squad. And lastly, a Virtual Interest Group (VIG) is similar to the “interest community” in Haggle. However, the construction of VIG is based on the similarity among the interest models.

3.4.1.1 Interest Learning and Forgetting

One of the interesting aspects of the CBMEN environment is the pace at which the situation changes, which requires that interests be learned and forgotten in a timely way. For the CBMEN project, we are using ARAMA, AIM with Reinforcement and Aging Modeling Algorithm (RAMA) to model the user’s interests. RAMA was developed in previous research programs and was demonstrated to be effective in a NIST evaluation (H. J. Li 2012) (R. P. Alonso 2010) (H. H.-A. Li 2009) (H. B. Li 2009) (R. a. Alonso, Model-Guided Information Discovery for Intelligence Analysis 2005) (R. a. Alonso, Combating Cognitive Biases in Information Retrieval 2005) (H. E. Li 2006) (R. B. Alonso 2003).

The RAMA algorithm adapts user interest models from user events by combining reinforcement learning with information aging. The algorithm processes user events differently based on their polarity. The polarity of a user event indicates whether it positively or negatively reflects user’s interests. Positive events express user’s interests and increase the importance of the topics contained in them. Negative events imply user’s disinterest and thus decrease the importance of the contained topics. For example, a subscription event is positive whereas a un-subscription event is negative. In this reinforcement scheme, different types of user events are assigned a different weight based on the extent to which the given type reflects the user’s interests. In addition, the weight assignment can be easily configured for the particular application. The active information aging process goes on continuously. The importance of an interesting topic in the past will gradually decrease and drop below the user’s radar screen unless reinforced positively over time. The modeling algorithm processes user events incrementally. This capability allows the RAMA to model users continuously and in real time. The scalability of the RAMA is also manifested in its ability to handle multiple concurrent users.

3.4.1.2 AIM2 Update

A user’s interests can be grouped into four categories: explicit, correlated, popular, and meta-interests. Explicit interests are intentionally expressed by the user via a query or subscription. They are captured in IA1 prototype. Correlated interests are inferred from content metadata or neighbor’s node description. These are linked or co-occurring with explicit interests. Here the term “linked” means new interest element involved in a RDF triple with explicit interest. Popular interests are inferred from content metadata or neighbor’s node description. They are frequent occurring interest elements in content metadata or neighbor’s node description. They capture sudden situational changes on
the ground. Given the reasoning capability of the OWL language, *meta-interests* can be derived purely from previous interests by reasoning based on the background ontology. For example, suppose the ontology states that A, B, C are all subclasses of D. If the node showed interests for A and B, it is likely that the node might also be interested in C.

Note that correlated, popular, and meta-interests are all *implicit interests* since they are not directly expressed by the user. Together with the explicit interests, they form a more complete picture of the user’s information needs. AIM2 supports correlated and popular interests in addition to explicit interests. The meta-interests are a feature in future release.

In Haggle content metadata and subscriptions are expressed as name-value pairs. AIM represents the interests extracted from these sources as elements of the Term class in an interest model. In order to support Drexel’s rich content metadata or rich subscription based on OWL/RDF, new classes are needed for ontological elements. The class OntologyNode extends the Element class and captures a node in an ontology. For example, the following are two ontology nodes:

```
<urn:swat:reports-20111031.owl#Report>
<urn:swat:reports-20111031.owl#SpotReport>
```

Another class addition is the OntologyRelation, which extends the Element class and captures an RDF triple. For example, an OntologyRelation may look as follows:

```
urn:swat:reports-20111031.owl:PositionReport
rdfs:subClassOf
```

In addition, it is useful to add two new members in the Element class: eventType and interestType. The former indicates the type of the source Haggle event (i.e. publication, subscription, or unsubscription) for the element, whereas the latter denotes the nature of the interests (i.e. explicit, correlated, popular or meta interests).

AIM2 has implemented the OntologyNode class and the eventType member. The other additions will be implemented in the next release.

In order to handle the rich metadata and rich queries, we have revised the RAMA algorithm for adapting user’s interest model. The pseudo-code for the revision is as follows.

1) Create an empty interest model for the local node if it does not exist.
2) Extract the interest elements from information associated with the user event.
   Assign appropriate interestType (explicit, correlated, or popular) to each element.
   - Haggle’s Attribute-value pairs
   - Convert to Term element
   - Drexel’s rich metadata

Distrion Statement A: Approved for Public Release, Distribution Unlimited
• Parse rich metadata into RDF triples (subject-predicate-object)
• Construct an OntologyNode for each node (subject/object)
• Construct an OntologyRelation for each triple

3) Age the current model by applying a forgetting function to all interest elements in the interest model.
4) If an information element from the event already exists in the model, its weight in the model is positively or negatively reinforced. The amount of reinforcement is modulated by the source event type.
5) Otherwise, if the user event is positive, insert new information elements from the event into the interest model with a default weight modulated by their source event type and interest type.

With the integration of Drexel’s registrar manager within the CBMEN system, the rich content metadata and rich query are specified as OWL/RDF documents written in Turtle syntax. In AIM2, these documents are passed using Raptor (version 2), the Raptor RDF syntax library (http://librdf.org/raptor/). This is the same library used by the Registrar Manager for handling OWL/RDF documents.

3.4.2 AIM Inputs and Outputs

The models maintained by AIM can be used by other CBMEN components. The inputs for AIM consist of multiple sources including the mission data, warfighter’s queries, battle drill profiles and the content generated or requested by the warfighter. With this information as input, AIM formulates its key output, namely an interest model.

AIM adapts to the MANET environment by updating interests and their weights based on node and network events (e.g., publish and subscribe events) that reflect node’s interests and information needs. More specifically, the ARAMA algorithm will process these events to adapt the interest model for the local node, which in turn will be used to construct its VIG. Most importantly for CBMEN, the interest model and the VIG can be used to influence opportunistic/proactive caching and prefetching decisions.
As depicted in Figure 11, at a high level the input for AIM will be the explicit interests (e.g. subscriptions/queries) and content attributes/metadata. The metadata can come from static content, like the mission related documents, and dynamic content, which are generated or circulated during a mission, such as a spot report. We do not rely on static content, but can take advantage of it if available.

The model adaptation algorithm (within Interest Modeling Manager, IMM) depicted in Figure 11 is an extension of the user modeling algorithm SAIC/SET previously developed and is based on reinforcement learning. The interest model representation has been initially implemented to work with Haggle's attribute-value pairs and subsequently adapted to rich metadata and queries utilizing OWL ontologies with the AIM2 update.

As shown in Figure 11, the primary output of AIM is an Interest Model. More specifically the output can be:

- Interest model for a node (Phase 1)
- Interest model for a group (may be considered in Phase 2)

Each interest model can be represented as an XML file (Table 1). The interest model is contained within the root node `<Models>`. An interest model may have one or more facets, which in turn may have one or more interest elements (i.e., terms). A term element has three attributes: identifier, weight, and text. This XML representation may be modified such that it can also be embedded in a node description to exchange with neighbors upon contacts.
<Models>
    <InterestModel identifier="Node#1">
        <Facet identifier="1" retired="0">
            <Term identifier="1" weight="0.8" name="armored vehicle" value="true"/>
            <Term identifier="2" weight="0.5" name="convoy" value="true"/>
        </Facet>
    </InterestModel>
</Models>

Table 1 XML Representation of a Sample Interest Model

3.4.2.1 AIM2 Update

AIM2 tracks and processes the following Haggle events to adapt the node’s interest model:

1) Local subscription and un-subscription. These events express the node user’s explicit interests. They are also being used in IA1 AIM prototype (i.e. AIM1).
2) Local content metadata publication. These events are used to derive correlated and popular interests. They are being processed in AIM2.
3) Content metadata publication by other nodes in the network. These events are also used to derive implicit interests. This is also a new feature in AIM2.

There are two formats for interests in the above events: name-value pairs (aka “plain attributes”) and OWL/RDF documents (aka “rich attributes”). AIM2 supports both formats. Table 2 shows an example of rich content metadata whereas Table 3 provides an example of a rich subscription.

@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix c2: <urn:swat:reports-20111031.owl#> .
@prefix wn30: <http://purl.org/vocabularies/princeton/wn30/> .
@prefix : <#> .

Table 2 An Example of Rich Content Metadata

With AIM2, a node’s interest model generated by the RAMA algorithm contains OntologyNode elements. Table 4 shows the interest model in XML format after processing the publication of the content metadata shown in Table 2. Table 5 shows the evolved interest model after RAM further processing the subscription depicted in Table 3.
{InterestModel: identifier=me, 
numReportedEvents=0, 
facets= 
[(OntologyNode: <urn:registrar:description#mIedY-wn>, 0.325000)] 
[(OntologyNode: <urn:swat:reports-20111031.owl#IEDAlertReport>, 0.325000)] 
[(OntologyNode: <http://purl.org/vocabularies/princeton/wn30/synset-loud-noun-1>, 0.325000)] 
[(OntologyNode: <http://purl.org/vocabularies/princeton/wn30/synset-smoke-noun-1>, 0.325000)] 
[(OntologyNode: <http://purl.org/vocabularies/princeton/wn30/synset-light-noun-1>, 0.325000)]}

Table 4 Interest Model with Rich Metadata after a Pub Event

{InterestModel: identifier=me, 
numReportedEvents=0, 
facets= 
[(OntologyNode: <urn:registrar:description#mIedY-wn>, 0.292500)] 
[(OntologyNode: <urn:swat:reports-20111031.owl#IEDAlertReport>, 0.650000)] 
[(OntologyNode: <http://purl.org/vocabularies/princeton/wn30/synset-loud-noun-1>, 0.292500)] 
[(OntologyNode: <http://purl.org/vocabularies/princeton/wn30/synset-smoke-noun-1>, 0.292500)] 
[(OntologyNode: <http://purl.org/vocabularies/princeton/wn30/synset-light-noun-1>, 0.292500)] 
[(OntologyNode: <urn:swat:reports-20111031.owl#PositionReport>, 0.650000)] 
[(OntologyNode: <urn:registrar:description#qIedX-wn>, 0.700000)]

Table 5 Interest Model with Rich Metadata after a Pub and then a Sub Event

3.4.3 AIM-Enabled Applications

In Phase 1, the basic interest model (and in Phase 2, its enabled capabilities) will be leveraged to support content management tailored to the warfighter’s information needs. See Figure 12 below. Note: Figure 12 depicts both Phase 1 and Phase 2 features. The ADM and Security Alerting as well as the VIG capability in the IMM are Phase 2 capabilities.

Figure 12 Applications Supported by the Interest Model

Opportunistic caching occurs when nodes in contact contain data objects that match the interest model or when data objects being forwarded match the interest model. These matching objects may be cached for future use even if the user has not explicitly requested them at this time. Proactive caching can take advantage of the interest model to retrieve relevant data objects on the node’s behalf automatically when the bandwidth
and other resources allow. This is called “prefetching” because contents are being cached for possible future use without user’s explicit request. In this case, the IM may also determine in what order the contents should be pre-fetched. In other words, the IM is important in prioritizing contents during prefetching when network resources are constrained.

In addition, IM can influence forwarding decisions because the resolution of targets for a data object is based on the matching of its metadata with the interests of all known remote nodes (Erik Nordstrom, A Search-based Network Architecture for Mobile Devices 2009). The interest model represents the normal behavior of a node in terms of what sort of information the node has been interested in seeking. Thus it is natural to use it to detect anomalies in node behavior (i.e., ADM to detect anomalous patterns of behavior and generate Security Alerts).

It is noteworthy that the AIM-based caching and replication services are naturally adaptive because AIM is continuous and adaptive. When the interest and weights in the IM change, the content cache needs to be updated by refreshing the content related to latest most important interests and reducing the importance annotation of contents that correspond to low interests.

For caching to be cooperative and replication to be proactive, the system needs to have a notion of the interests of neighboring nodes. This can be achieved in three ways: 1) by exchanging their IMs locally with their neighbors (Phase 1); or 2) by creating the IMs at the level of a cluster using AIM (Phase 2) and 3) by creating VIG-based Interest Models (Phase 2). Refreshing and purging can be performed similarly to the above process. In the case of replication, the number of copies of content should correlate with the amount of common interests of the neighborhood. Of course, the extent of replication should also consider the node mobility and link status. Important content needs to be replicated more often if nodes are highly mobile and/or links are very unreliable. There are two ways that interest models can be shared amongst different nodes. One way is to embed the model as part of the node description, which gets exchanged with neighbors upon contacts. The model can also be shared as an independent data object, like rich metadata or rich queries.

In the future, the Interest Models and their enabled capabilities can also be utilized by network services for smarter routing decisions. For example: based on the IM, dynamic queries can be automatically generated (R. a. Alonso, Combating Cognitive Biases in Information Retrieval 2005) for pre-fetching contents that are most relevant. For example, if our interest model derives some interest X based on the fact that X frequently co-occurs in published content’s metadata with node’s registered interests A and B, the system may pre-fetch content based on interest X for the node. If we have domain knowledge about the user’s roles or battle drills, there is another interesting type of pre-fetching that can be performed as we describe next.

### 3.4.3.1 IM-Enabled and Profile-Based Pre-Fetching

An IM-Enabled and Profile-Based Pre-Fetching service (IPP) for IE1 has been developed. IPP requires that we have domain knowledge about the user’s roles and/or battle drills. At the high level, the adaptive interest model is used to select the best matches from a pool of pre-defined static information profiles created based on the roles
or battle drills. The IMM does this using the Interest Models and the Profiles. The best-matched profiles are then used by the CBMEN system to proactively pre-fetch any data objects that fit the profile (see Figure 1)

![Figure 13 IM-Enabled and Profile-Based Prefetching](image)

```json
{InterestModel:identifier='IED',
 Profile,numReportedEvents=1,
 facet1={facet; ID=1; retired=0; Terms=['[Picture_armoredvehicle, 1]', '[Picture_convoy, 1]', '[Picture_explosive, 1]', '[Picture_explosion, 1]', '[Picture_wires, 1]', '[Picture_ordnance, 1]', '[Picture_smoke, 1]', '[Picture_light, 1]', '[Picture_wounded, 1]', '[Picture_bomb, 1]']}
```

![Figure 14 Example of a profile based on the battle drill for IED](image)

The large blue eclipses represent static information profiles whereas the red dots represent the evolving IM for the node. The blue curve represents the trajectory of the IM.

For the IPP service, we first build a selective set of information profiles using domain knowledge. For example, we can create information profiles for Cordon, Search, and IED using battle drill descriptions\(^1\). The information profile uses the same data structure

\(^1\) According to Sergeant’s Time Training.COM, a battle drill is defined as follows: “A battle drill is a collective action executed by a platoon or smaller element without the
as IM. In the example shown in Figure 14 the IED profile contains one facet, which in turn consists of ten terms that capture information needs related to the IED battle drill. AIM builds and continuously adapts the IM based on the events reflecting the node’s interests such as subscription, un-subscription, and content publication.

Triggered by new interest subscription and unsubscription events, the IM is compared with the profiles to find the best match for prefetching. The match is determined by the cosine similarity between two vectors. As the IM continues to adapt and change over time, the current profile may become dissimilar and a new profile may become more similar. This can happen when the situation on the ground changes. For example, cordon and search operation is interrupted by an IED explosion. In this case, the IM may initially match cordon and search profiles, but then transitions to an IED profile.

When a match is found, the information elements or interests of the profile that are not covered by the IM will be used for prefetching. With Haggle, this can be achieved by registering these new interests via a virtual application. These interests are then included in the node description that is exchanged with neighbors. Content matching these interests is automatically prefetched and cached when it become available and typically before it is requested by the user.

3.4.3.2 AIM2 Update

With AIM1, prefetching is based on interest model & battle drill profiles. With AIM Version 2 (short AIM2) and beyond, there are two possibilities below.

1) Extend AIM1 application by replacing attribute-value pairs with Drexel rich metadata. This is implemented in AIM2.
2) Instead of using profiles, new prefetching applications can be based on derived interests (correlated or popular interest). This type of applications can be developed for future releases. Queries can be generated automatically using derived interests and fetch contents proactively. The steps involved in such an application are described below.

   a. Use published content metadata to derive interests and evolve the interest model.
   b. Automatically generate a rich query with top derived interests from the interest model.
   c. Submit the generated query to the Registrar on the node’s behalf.
   d. Content with metadata matching the query will be fetched by the node and stored in cache.

An example of a profile described in rich metadata is shown Figure 15. Note that this RDF document contains a number of ontological elements from WordNet 3.0 in RDF\(^2\) to complement the ontology developed by Drexel for IA1.

---

\(^2\) Freely available at http://eculture.cs.vu.nl/git/public/?p=vocs/wordnet.git;a=tree;f=rdf;hb=HEAD.
AIM2 supports the following three scenarios:

1) IPP with rich profiles only. The interest model is adapted with both subscription and publication events, moving close to a search profile first, then to an IED profile, and lastly to a cordon profile, and activate the profile as a result. The activation of the profile leads to the launching of a preloaded rich query, which prefetches contents when they're published and being matched.

2) IPP with plain profiles only.

3) IPP with mixed both plain and rich profiles.

3.4.3.3 Explicit Profile-Based Pre-Fetching

A profile may be explicitly activated independently of the interest model. Such a profile is called a mission interest profile. To activate mission support, make sure the tag_field and tag_field_value configuration parameters are set. Then post a data object with the attribute-value pair tag_field=tag_field_value, a mission=<mission name> attribute-value pair, and attach an XML file containing a list of interest manager profiles that capture interests for content relevant to the mission that should be explicitly prefetched.
Any Haggle instance that subscribes to "mission=<mission name>" will then subscribe to all terms listed in the profile, independent of any limitations on profile or interest subscriptions set for regular profiles. Posting a new data object with the same mission name and a different profile (including the possibility of an empty profile) will overwrite the existing profile assignment for that mission.

3.4.4 Interest Model Sharing

We have implemented IM sharing capability in AIM Version 3 (short AIM 3) with the following features:

1) Unit-based sharing controlled by the configuration parameter "units". When sending out an IM to neighbors for sharing, the data object will carry the information regarding unit as the value of the IM sharing attribute. This attribute takes the following form:

   IMS_ATTR_NAME=alpha,

where alpha is a unit name.

Upon receiving an IM from a neighbor, the local node's units info will be compared to that carried on the incoming IM data object. If matching occurs, the incoming IM will be processed by the Interest Manager. Otherwise it will be ignored.

2) The shared IM is incorporated into receiving node's IM. When the Interest Manager receives an incoming data object, it checks to see if it carries the IM sharing attribute (i.e., IMS_ATTR_NAME=alpha). If positive, the shared IM elements will be extracted from the data object and send to adaptation algorithm to update the local IM.

3) IM sharing triggered by significant change in the IM. When the change (defined as 1 minus the similarity between the current IM and the state of the IM when it was last sent) is greater than the change threshold, defined in the configuration file, the IM will be sent to all connected neighbors.

4) The latest version of AIM3 works together with Drexel Registrar and is fully compatible with AIM2 prefetching.

3.4.4.1 IM Sharing Sample Scenarios

We describe two IM sharing based scenarios below.

1) Situational awareness (SA) is critical in operational environments. SA can be enhanced by knowing what other edge network users already know. IM sharing allows mobile users to enhance SA by dynamically building the common operating picture at the edge as illustrated in the figure below.
2) The sooner a new user can be given necessary information, the sooner everyone can get back to focusing on achieving objectives. IM sharing allows new users to be quickly brought up to speed by learning from the experience of their peers. This is illustrated in the following figure.

3.4.5 **AIM Implementation and Haggle Integration**

The AIM components have been integrated into CBMEN Haggle architecture as part of a new interest model manager. The interest model built by AIM is implemented as a data structure that by means of an internal interface can provide interest-model-based services to Haggle modules and managers.

The interest model manager includes the following functionality:
• ARAMA interest model creation and maintenance
• Interest profile management, parsing, and configuration
• Interest model classification
• Prefetching using a virtual application

The interest model manager listens for subscribe/unsubscribe events by registering for the same IPC events as the application manager uses to process subscriptions (see InterestManager::onReceiveFromApplication()). Pre-fetching is implemented by inserting a virtual application node into the data store with the interests the interest model manager wants to pre-fetch based on its interest model classification, and then triggering a reconstruction of the node description to include those interests and broadcasting it on the network (see InterestManager::updateInterests()).

3.4.5.1 AIM2 Interest Manager Update

The latest version works with rich metadata and performs prefetching through Drexel's registrar manager. It has retained the IE1 functionality to work with native Haggle attributes as well. At runtime, AIM2's InterestManager adapts the interest model using subscription and publication events. Using an example similar to IE1, three profiles are used for prefetching: search, ied, and cordon, which are specified in the configuration file. These profiles are encoded in Turtle RDF. When the interest model is updated, it is compared to all three profiles. If the similarity of a profile exceeds predefined threshold specified in the configuration file, the profile is said to be activated and a preloaded rich query (specified in Turtle RDF) will be submitted automatically by the InterestManager. When a publication's metadata matches the preloaded query, the related content will be fetched without user's explicit request. The metadata matching with the rich query is a service provided by Drexel's registrar manager.

The interest manager listens for rich metadata publication and subscription events by filtering all incoming data objects with special attributes signal rich metadata (see InterestManager::onIncomingDataObject(Event *e)). The rich metadata (OWL/RDF documents in Turtle) are processed and RDF triples are being parsed using the raptor2 library (see ar/aimrdf.h and ar/aimrdf.c).

The parsed triples are processed by the AIM model adaptation algorithm to update the node's interest model. The model now has a new class of elements: OntologyNode to represent interests on the ontological elements (e.g. subject and object in a RDF triple). The significance of an ontology element in a triple is determined by two factors:

(i) the nature of the source event, i.e., publication vs. subscription (see Adaptation::getEventRelevance(int eventType)), and
(ii) the type of the predicate, i.e., subclass, is-a, or intersection (see InterestManager::computeWeight(const char *predicate, int *weight)).
Modified Classes

DebugManager: The DebugManager has been modified to be able to send debug messages to the Interest Manager, prompting it to print debug information.

New Classes

InterestManager: The interest manager mediates between Haggle applications and the Adaptation module, and implements a pre-fetching system based on a virtual application. It registers for events that are generated when an application subscribes or unsubscribes to a topic, and feeds those events into the Adaptation class, which produces a user model. The interest manager then picks a representative selection of interests from a set of pre-defined interest profiles (which may be distinct from the set of interests explicitly registered) and registers a virtual application with those interests.

The virtual application causes those interests to appear in the node description and propagate to other nodes, which send back relevant content. When the user or application becomes interested in a pre-fetched piece of content, it appears immediately, as it is already in the local data store.

Adaptation: Implements the ARAMA algorithm that learns the interest models. It is fed a stream of subscription and unsubscription requests and produces an interest model based on the adaptive algorithm.

InterestModel: Represents the learned interests for a node. An interest model represents a cohesive set of things that a user or application may be interested in. It is
comprised of several Facets, and may be compared to other interest models using a cosine similarity metric.

**Facet:** Describes one aspect or facet of a user's interests. That facet is represented by a weighted set of Terms, each corresponding to a particular term, pair of terms, or topic area. Apart from describing the facet itself, this class also records the temporal intervals during which the user's attention was devoted to that facet of their interests. An interest model may contain several facets.

**Element:** Represents some dimension of node's interest, with an associated weight indicating the level of interest. This is the superclass for several types of elements including terms, named entities, topics, and relationships. Currently only the term element is implemented. A facet contains one or more elements.

**Term:** Term derives from Element and represents a user interest dimension corresponding to a word or phrase. The word or phrase may, for example, be a simple word or named entity (noun phrase), selected based on its relative frequency in an application's subscription requests.

**OntologyNode:** Similar to Term this class derives from Element and represents a user interest dimension corresponding to element of an ontology.

**AimAttribute:** Represents a Haggle attribute in the form of a name-value pair. The AimAttribute class encapsulates all the information the Adaptation needs about a Haggle attribute in order to update the interest model. It is used only as a means of information exchange between the interest manager and Adaptation classes.

### 3.4.6 Sample Configuration and Parameters

The interest model manager is configured by the `<InterestManager>` tag in the configuration file. An excerpt of a configuration that uses adaptive interest modeling to activate the best-matching profiles from a set of preloaded interest profiles is shown below:

```xml
<InterestManager
    enabled="true"
    prefetch="true"
    similarityThreshold="0.2"
    maxInitialWeight="0.6"
    reinforcementFactor="0.3"
    agingFactor="0.10"
    maxInterests="6"
    maxProfiles="2">
    <Profile filename="/tmp/demo_profiles.xml" />
</InterestManager>
```

The parameters are defined as follows:
enabled - If this attribute is absent or set to any value except "true", the Interest Manager will be completely deactivated and will unregister itself from all events it would normally receive.

prefetch - If this attribute is absent or set to any value except "true", the Interest Manager will build interest models as normal, but will not use those models to prefetch relevant data objects.

maxProfiles - an integer. Specifies the maximum number of profiles that can be said to be matching the current user model. If set to 1, then only the best profile will be used for pre-fetching. If set to N, the best N profiles will be used. If unspecified or zero, then unlimited profiles may match.

maxInterests - an integer. Specifies the maximum number of interests to pre-fetch. If unspecified or zero, then all interests from matched profiles will be pre-fetched. If set to N, then the N top interests from all matching profiles will be used (where "top" is determined by the "weight" value each interest is assigned by its profile).

similarityThreshold - a floating point number between 0 and 1. Specifies the minimum similarity at which an information profile can be said to be matching the user model. Set it to 0 to match the best profile regardless of absolute similarity.

maxInitialWeight - a floating point number between 0 and 1. Specifies the approximate starting weight of terms introduced into the interest model.

reinforcementFactor - a floating point number between 0 and 1. Specifies how much of an impact new subscriptions/unsubscription should have on the interest model. Higher values lead to quicker adaptation, lower values slower.

agingFactor - a floating point number between 0 and 1. Specifies how quickly terms that haven't been subscribed to in awhile decay in relevance. Higher numbers mean faster decay.

The Profile tag (of which there may be more than one) links to an XML file describing a set of information profiles against which the user model is matched.

A similar configuration that uses rich interest profiles (in Turtle syntax) with preloaded rich queries (again in Turtle) associated with each of them can be specified as follows:
3.4.6.1 Mission Options

An excerpt of a configuration that explicitly activates mission profiles that are published as ordinary data objects by another node, e.g. the squad leader, is shown below:

```
<InterestManager
   enabled="true"
   prefetch="true"
   similarityThreshold="0.2"
   maxInitialWeight="0.6"
   reinforcementFactor="0.3"
   agingFactor="0.10"
   maxInterests="6"
   maxProfiles="2">
   <RichProfile filename=file:///tmp/searchProfile.ttl queryFilePath="/tmp/qSearchX.ttl" />
   <RichProfile filename=file:///tmp/iedProfile.ttl queryFilePath="/tmp/qledX-wn.ttl" />
   <RichProfile filename=file:///tmp/cordonProfile.ttl queryFilePath="/tmp/qCordonX.ttl" />
</InterestManager>
```

The new parameters are defined as follows:

- **tag_field, tag_field_value** - strings. These attributes control the interest manager's mission profile assignment support. If tag_field and tag_field_value are both set then the Interest Manager will examine all incoming data objects which have the attribute-value pair tag_field=tag_field_value. If such objects additionally have a mission=<mission name> attribute and an attached XML file, they will be considered as mission profile assignments.

- **mission_time_tag** - a string. Specifies where the Interest Manager should look to determine the creation time of a mission profile. If set, it will look at the value of the attribute specified. If unset, the data object's creation time field will be used instead.
3.4.6.2 Rich Metadata Control Options

AIM3 added two options ("richMetadataAware" and "richMetadataLocalOnly") to control the processing of rich metadata in the configuration for Interest Manager as follows:

```xml
<InterestManager
    enabled="true"
    richMetadataAware="true"
    richMetadataLocalOnly="false"
    prefetch="true"
    debug="3"
    similarityThreshold="0.2"
    maxInitialWeight="0.6"
    reinforcementFactor="0.3"
    agingFactor="0.10"
    maxInterests="6">
</InterestManager>
```

*richMetadataAware* - if it is set to true, rich metadata will be processed. Otherwise it will be ignored. The default is true.

*richMetadataLocalOnly* - further refines the richMetadataAware parameter to restrict processing to local rich metadata only. The default is true.

3.4.6.3 Interest Model Sharing Options

The interest model sharing can be configured as below:

```xml
<InterestManager
    <InterestModelSharing
        enabled="true"
        units="alpha"
        limit="5"
        contactTrigger="true"
        changeTrigger="true"
        changeThreshold="0.42" />
</InterestManager>
```

The new parameters are defined as follows:

*enabled* - turns on or off IM sharing capability.

*limit* - indicates how many top-weighted elements of the IM are to be shared.

*units* - specifies the unit name with whom IM sharing is allowed.

*contactTrigger* - enable node contact as a trigger of IM sharing.

*changeTrigger* - enable significant IM change as a trigger of IM sharing.

*changeThreshold* - specifies the amount of change in IM to trigger sharing.
3.4.7 Limitations and Possible Future Directions

One limitation of the current interest model manager implementation is that it only takes into account subscribe/unsubscribe events. A natural extension could take into account publish events, assuming that the attributes of published data objects are correlated to the node’s own interest.

The initial approach to interest modeling closely matched the paradigm of unmodified Haggle, which associates sets of interests (represented as attribute-value pairs) with nodes. A set in this context has a disjunctive interpretation. It remains to be investigated how more complex interests, e.g. rich queries, can become inputs and outputs of the interest model manager. The currently implemented approach as part of the AIM2 update is to directly support rich metadata and queries, which can support conjunctions and more complex interests. Another possible approach may involve abstractions, e.g. mappings from rich representations into attribute-value pairs.

In summary, the interest model enables a range of capabilities. These include:

- Automatically generating interest for content retrieval (already implemented)
- Ranking content relevance/utility (preliminary experiments planned for Phase 1)
- Building a virtual interest group (preliminary experiments planned for Phase 1)
- Prioritizing a list of content
- Relevant content recommendations
- User query contextualizing and modification
- Detection of anomalies (Phase 2)

Depending on the CBMEN target scenarios, the capabilities of the CBMEN API, and the resource requirements, some of these capabilities, if not already planned, might be worthwhile to investigate as the program evolves.

Also, as indicated before, the exchange of interest models and building interest models for groups rather than individual nodes seems feasible, but the benefits and implications for networking resources need to be carefully examined. General mechanisms to limit the amount of overhead created for other nodes (e.g. limiting automatically generated interests) in the network are especially important to consider.

There are a couple of near-term directions we can pursue. Firstly, in the prefetching application described above, allow profiles to be dynamically loaded during a mission in addition to loading at startup. This is very useful in operational settings where the ground situation can change very quickly and unexpectedly. As a result, new profiles may be necessary to accommodate such unplanned changes.

Secondly, related to the first direction, we should also allow the preloaded query to be dynamically loaded during the mission. The reason is the same as that for the first direction.

Lastly, instead of prefetching based on preloaded queries, the interest manager could automatically generate queries at runtime based on the latest interest model. This capability complements the last two directions in that it will prefetch information based
on latest information need without requiring the manual effort in produce and loading profiles and profile-specific queries.
3.5 Network Coding


3.5.1 Network Coding Approach

Network coding is known to be beneficial in wireless and mobile ad-hoc networks in various applications due to its ability of best utilizing multipath and partial transmissions. The basic random network coding concept is to perform coding operations among content blocks at each participating node. By receiving enough linearly independent coded blocks from any participants, the content receiver is able to recover the original content. Network coding has been proven to be sufficient to achieve the maximum capacity of a multicast session (Ahlswede, 2000) in a lossless network. In addition, network coding is a natural remedy of partial, unsuccessful transmissions over high loss channels and thus is helpful for improving wireless communication reliability. The major reason of these benefits is due to the information mixing nature of network coding. By mixing information within a generation of content blocks, each network coded block has the same degree of fresh information, no matter which path or which encoder it is from. Thus, network coding solves many issues in multipath transmissions and efficiently exploits previously received partial data object. With the advantages that network coding provides, this design document examines how Haggle can be enhanced to support network coding in MANET environments.

Our approach is based on an adaptation of Code-Torrent (U. Lee 2006) to content-based networking with content caching and interest-driven content-dissemination. Figure 19 illustrates the main idea. Assume content (e.g., a file) is already cached in multiple nodes of the network (three sources in this scenario). Once the content is requested, i.e. the interest matches the content attributes, network coding is invoked to split the content into blocks (three in this example) and code them into a single block, which is a random linear combination. In general, each source sends a stream of such blocks. The random coefficients are always new so that with high probability the same block is never generated again and a receiver will receive new information (an innovative block, i.e. a block that is linearly independent of previous blocks) with high probability. Hence, in our example the receiver only needs three blocks to decode (solving a system of linear equations) and reconstruct the original file. It does not matter where these blocks come from, how they were routed, or if there was a temporary disruption. Different from traditional fragmentation each block is as good as every other one and has additional information content.
Haggle is centered on the notion of a data object, which is composed of the attributes and the plaintext (or encrypted) content of the objects. Our network coding implementation will not encode the attributes but only the potentially large content part of the data object. The attributes are kept unencoded, because Haggle needs to resolve interest against data objects for the purpose of content-based forwarding to peers.

There are some challenges that must be overcome with the implementation of network coding in Haggle. The Haggle framework operates under the assumption that a data object is a valid and fully received object (there are no partial data objects) and thus any components inside the framework operate with such an expectation. Haggle transmissions operate in a transaction mode where an object is sent and received in an all or none manner. Unmodified Haggle does not support partial transmissions, resuming fetching an object, or multipath transmissions, which we believe are essential features for network coding to be beneficial. Finally, since the network coding protocol exchanges coded blocks, there must be a mechanism to store and forward these blocks at intermediate nodes.

This design document explains how network coding is integrated into Haggle and how it will operate by representing blocks, which are generated by the encoder, as special data objects. Block data objects are created on demand when a normal data objects needs to be sent, without being inserted into the data store at the sender node (and

Figure 19 Network Coding in a Simple Scenario

3.5.2 Network Coding Implementation and Haggle Integration
thus avoiding unnecessary data base overhead at the sender). At the receiver and potential intermediate nodes, however they are treated as full featured data objects, that is they enter the data store until they are purged, e.g. in the case where sufficiently many innovative blocks have been received to recreate the original data objects. This design should allow us to experiment with forwarding of network coded blocks through intermediate nodes and should also allow potential receivers to collect blocks from multiple sources, which is essential to exploit the benefit of network coding in mobile networks.

Network coding is implemented in Haggle as a standalone manager. This allows it to be turned on or off depending on configuration and dynamically based on the type of traffic.

Haggle natively supports associating attributes with a data object, which we leverage in the implementation. Block data objects inherit all attributes of the data object that have been generated from and include several additional attributes as mentioned below.

**New Attributes for Block Data Objects**

We include additional attributes in all block data objects to provide information about the parent data object the block was derived from:

- \_NC\_ORIG\_ID\_: The data object id of the original data object
- \_NC\_ORIG\_DATA\_LEN\_: The length of the parent data object content
- \_NC\_ORIG\_FILE\_NAME\_: The file name of the parent data object
- \_NC\_ORIG\_CREATE\_TIME\_: The create time of the parent data object
- \_NC\_ORIG\_SIGNEE\_: The signee of the parent data object
- \_NC\_ORIG\_SIGNATURE\_: The signature of the parent data object

Our implementation is slightly more general than explained above in that for experimental purposes we allow multiple blocks to be included in a single network-coded block data object, but allow them all to share the same header. For sake of simplicity, we abstract from that level of detail in the following.

**New Network Coding Event Flow**

The network coding event flow in Figure 21 shows the new flow links red contrasted with the original event flow in Figure 20. Green represents the send events, which contain network coded block data objects, node descriptions, or control messages. The purple event is the translation of the network coded block data object to the original data object for processing by the forwarding manager.
ProtocolManager

- **EVENT_TYPE_DATAOBJECT_SEND**
  A new check is performed to see if the content object should be network coded. If so, **EVENT_TYPE_DATAOBJECT_SENDNETWORKCODING** is generated to trigger network-coded blocks to be generated and sent.

ForwardingManager

- **EVENT_TYPE_DATAOBJECT_SENDNETWORKCODING_SUCCESSFUL**
  In the method onSendDataObjectResult, the forwarding manager will raise event **EVENT_TYPE_DATAOBJECT_SENDNETWORKCODING** for the original content data object so that the process for sending network-coded blocks can be repeated.
NetworkCodingManager

- **EVENT_TYPE_DATAOBJECT_SEND_NETWORKCODING**
  In the method `onDataObjectForSendNetworkCodedBlock`, the network coding manager reads the data object content and generates a single block. The number of network coded blocks may be adjusted as we experiment and measure for optimal sizes. The block is then added to the event queue by means of `EVENT_TYPE_DATAOBJECT_SEND`.

- **EVENT_TYPE_DATAOBJECT_RECEIVED, EVENT_TYPE_DATAOBJECT_NEW**
  The method `onDataObjectForReceiveNetworkCodedBlock` is called if the protocol manager generates a `EVENT_TYPE_DATAOBJECT_RECEIVED` event when it has fully created the data object. Both the network coding manager and security manager listen for the received event. This allows the received block data object to be passed on to the data store and then to the forwarding manager so that other nodes may receive the block. In parallel, the network coding manager also listens for the `EVENT_TYPE_DATAOBJECT_NEW` event so that it can gather all the network coded blocks (after verification by the security and data managers) for a data object so that once it receives enough innovative packets the blocks can be decoded to create the data object. Once the original data object is decoded and created, the event `EVENT_TYPE_DATAOBJECT_RECEIVED` is generated by the network coding manager (which will trigger verification of the reconstructed content by the security and data managers). An endless event creation cycle is avoided, because the network coding manager returns immediately if the data object is not a block by checking for the ORIGID attribute.

- **EVENT_TYPE_DATAOBJECT_SEND_NETWORKCODING_SUCCESSFUL, EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL**
  The method `onDataObjectSendSuccessful` intercepts the `EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL` event generated by protocol manager. It checks if the content object has been network coded. If so it retrieves the corresponding original content object and raises `EVENT_TYPE_DATAOBJECT_SEND_NETWORKCODING_SUCCESSFUL` for processing by the forwarding manager.

- **EVENT_TYPE_DATAOBJECT_SEND_FAILURE**
  In a way similar to the success case, this event, if raised for a coded block, will be passed through to the forwarding manager as an `EVENT_TYPE_DATAOBJECT_SEND_FAILURE` event for the original data object.

New Classes

**NetworkCodingManager:** Maintains suitable instances of NetworkCodingEncoder and NetworkCodingEncoder on a per-data-object basis. Listens for events `EVENT_TYPE_DATAOBJECT_SEND_NETWORKCODING, EVENT_TYPE_DATAOBJECT_RECEIVED, EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL`, and invokes encoder and decoder as needed. A manager module is created for both the network coding encoder and network coding decoder so a separate thread can be used for each task respectively.
**NetworkCodingEncoder:** Generates a coded block from the given data object. May be configured to generate more coded blocks in the future based on experiments and measurements. Generates `EVENT_TYPE_DATAOBJECT_SEND` once the coded block is generated.

**NetworkCodingDecoder:** Gathers all coded blocks until enough innovative blocks are received to generate the data object. Generates `EVENT_TYPE_DATAOBJECT_RECEIVED` for the generated data object.

Both the encoder and the decoder use the existing Code-Torrent classes to perform the actual coding and decoding operations.

**Modified Classes**

**ProtocolManager:** Added if statement to check if `EVENT_TYPE_DATAOBJECT_SEND` should generate `EVENT_TYPE_DATAOBJECT_SEND_NETWORKCODING`. Control message and node descriptions are never selected for coding. Selective/adaptive and more general content-based application of network coding could also be implemented here.

**ForwardingManager:** `EVENT_TYPE_DATAOBJECT_SEND_NETWORKCODING_SUCCESSFUL` is treated as `EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL`, but the data manager is not inserting the original data object into the Bloom filter of the peer, because it signals only partial success. We don’t know yet that the original data object has been fully reconstructed at the receiver.

**Event:** Existing code has if statements checking for `EVENT_TYPE_DATAOBJECT_SEND` and perform work such as adding the node to the node list. The treatment of `EVENT_TYPE_DATAOBJECT_SEND_NETWORKCODING` has been added to the same if statements.

**Protocol:** A performance optimization where the receiver uses a variant of Haggle’s REJECT message (called REJECT2) if the original data object has been reconstructed (which can be detected by a Bloom filter check) has been implemented here. The differentiation between REJECT and REJECT2 is needed to convey to the sender that the reason for reject is not the data object sent but its parent object, and hence no further blocks are needed. This is short-circuiting the normal propagation of the Bloom filter to the sender for the one-hop case. The sender will raise a `EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL` for the last block in this case, which the network coding manager translates into the same event for the original data object, which then is inserted into the peer’s Bloom filter by the data manager.
3.5.3 Sample Configuration and Parameters

A sample configuration file excerpt to enable the network coding manager is shown below.

```xml
<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** - false disables network coding no matter the content or context.

- **enable_forwarding** - 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** – retransmit 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** - the minimum file size for the data object to be eligible for network coding

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

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

- **resend_delay** – this delay (in seconds) determines the rate at which a new network coded blocks are generated and sent.

- **resend_reconstructed_delay** - the delay (in seconds) for sending (and hence reencoding) reconstructed data objects at intermediate hops

- **delay_delete_networkcodedblocks** – the delay (in seconds) after which network-coded block data objects are deleted at the receiver and intermediate hops
delay_delete_reconstructed_networkcodedblocks – the delay (in seconds) after which network-coded block data objects 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 - maximum age of network-coded blocks (in seconds) in the caching layer at the encoder after the last send event before being discarded.

max_age_decoder - maximum age of network decoder state.

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 selective network coding:

```xml
<NetworkCodingManager enable_network_coding="true"
  source_encoding_whitelist="n1,n2,n3" target_encoding_whitelist="n5,n7,n8"
  enable_forwardings="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 - comma separated values of node names which are allowed to encode content.

target_encoding_whitelist - comma separated values of node names which are allowed to decode content.

Since network coding is computationally expensive 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:

```xml
<NetworkCodingManager enable_network_coding="true"
  enable_forwardings="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>
```
3.5.4 Limitations and Possible Future Directions

**Efficiency of Network Coding** Haggle supports a single data object container concept without support for fragmentation or blocks, which makes it difficult to support multipath transmissions, resuming of transmissions, and efficient forwarding for network coding. This design document adopts the solution to represent blocks as ordinary data objects, which can lead to performance issues because each data objects is stored and forwarded individually. A form of aggregation at intermediate nodes might help, which would also enable the mixing of blocks that have been aggregated. Furthermore, for the first iteration the design uses a single generation for an object and a fixed generation size, which limits the content size to which network coding can be applied. Thanks to fragmentation (see next section), a large data objects can be fragmented into multiple generations (as fragments are called in the context of network coding), which can be transferred concurrently. Subsequent iterations may explore variable generation size, and the rate to generate block data objects for better information diversity and performance. Possible approaches to improve wireless communication reliability may also be explored by studying how the network coding parameters effects in blocks delivery process. We believe that the performance of network coding needs to be studied in the context of unreliable protocols and broadcasting, e.g. utilizing UDP or NORM.

**Countermeasures against Pollution Attacks** Since Haggle creates a hash signature composed of the data object’s attributes and contents, pollution attacks are possible if the relays are required to mix packets. To reduce the vulnerability to network coding pollution attacks, only the source node will encode the blocks. Relays will simply forward block data objects from their local store. Subsequent designs may examine the tradeoffs of allowing the relays to mix already encoded block data objects. There has been recent research on homomorphic signatures for network coding, which provides some solutions, but the computational overhead is significant need to be carefully examined to understand if the techniques are suitable for our resource-constrained Android target platform. Less expensive alternatives based on identifying the source of pollution and taking suitable countermeasures seem preferable. Our current approach uses relative large block sizes (e.g. 32K) that not only leads to a better amortization for content-processing overhead, but also has the advantage that the number of blocks is...
generally small enough so that they can be individually signed and verified with reasonable overhead.

*Reusing Computations* Images may be edited and transcoded for multiple screen sizes, which can be seen as another form of coding on top of a network coding protocol. This raises the question of how to efficiently transmit, fetch, and store these data objects. For instance, if an image of a map is edited, the resulting content may be the original image, the edited image for the screen size it was edited on, and the diff that describes the transformation applied to the image. Nodes have the option of retrieving the edited image and transcoding, or fetching the original image and the diff and transcoding for their screen size. Finally, this resulting image is stored as content as the transcoded image and the diff. As a result not only can we take advantage of network coding’s linear independent segments, though we can reduce the necessary content needed to fetch the latest version of an image for a particular screen size. For CBMEN it might be worthwhile to investigate if significant bandwidth reductions are possible using such an approach, which seems move the idea of network coding to a higher level of abstraction.
3.6 Fragmentation

Fragmentation provides an alternative approach to split a large object into smaller segments taking advantage of multiple source distribution and partial object transmission. As compared to network coding fragmentation does have an increased overhead in terms of disseminating the information to agree on which segments to transfer and which segments a receiver may already have. The good news is that this information is already maintained by Haggle by means of Bloom filters and disseminated as part of the node descriptions (at least in a delayed fashion). On the other hand, fragmentation does not have the computational overhead associated with network coding operations, which is why it is a good comparison point for performance evaluation. Furthermore, if the number of blocks needed for network coding becomes large, the header overhead for the coding coefficients can become very significant. Hence, in practice it is necessary to combine fragmentation and network coding for transmitting a very large data object. In this hybrid scheme, that our implementation also supports, the fragments are usually referred to as generations in the network coding literature.

3.6.1 Fragmentation Implementation and Haggle Integration

Fragmenting files is a well-known and indispensable mechanism in peer-to-peer networking. Although the content-based approach and the interaction between fragmentation and the other CBMEN features is interesting by itself, the discussion in the following will focus on the differences in implementation of fragmentation versus network coding in CBMEN. Fragmentation closely follows the event flow introduced by network coding in the previous section. The main difference is the event names use the term FRAGMENTATION instead of NETWORKCODING.

Due to the receiver needing every unique segment in order to reconstruct the parent object, our approach is what we call informed randomized fragmentation. The Bloom filter knowledge exchange is leveraged to determine which segments the receiver already has so that the sender can send only the missing fragments. The missing fragments are shuffled at the sender so as to increase the diversity in a MANET environment where there may be multiple concurrently contributing sources for a single data object, channel loss may occur, and disconnects will happen. Randomization also helps if the Bloom filter information is not fully up-to-date as typical in these environments.

New Attributes for Fragmented Data Objects

Very similar to network coding, we include additional attributes in all block data objects to provide information about the parent data object the block was derived from:

- _FR.ORIG_ID_: The data object id of the original data object
- _FR_SEQ_NUM_: The sequence number this segment represents
- _FR.ORIG_DATA_LEN_: The length of the parent data object content
- _FR.ORIG_FILE_NAME_: The file name of the parent data object
● _FR_ORIG_CREATE_TIME_: The create time of the parent data object
● _FR_ORIG_SIGNEE_: The signee of the parent data object
● _FR_ORIG_SIGNATURE_: The signature of the parent data object

The key difference from network coding is that fragments are equipped with a sequence number _FR_SEQ_NUM_. Our implementation is slightly more general in that for experimental purposes it allows multiple fragments to be included in a single fragment data object, giving rise to a list of sequence numbers, but for sake of simplicity we abstract from that level of detail.

**New Fragmentation Event Flow**

The event flow below is very similar to network coding, but apart from some subtle differences, there is a fundamental difference in when the fragmentation takes place. Network coding is performed on the fly block by block yielding a potentially infinite stream, but fragmentation is done once for each data objects into a finite sequence of fragments. Also fragmentation uses randomization and knowledge about missing fragments, where is network coding is already inherently randomized and blocks are never sent again with high probability.

**ProtocolManager**
- **EVENT_TYPE_DATAOBJECT_SEND**
  A new check is performed to see if the content object should be fragmented. If so, **EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION** is generated to trigger fragmented blocks to be generated and sent.

**ForwardingManager**
- **EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION_SUCCESSFUL**
  In the method onSendDataObjectResult, the forwarding manager will raise event **EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION** for the original content data object so that the process for sending fragments can be repeated.

**FragmentationManager**
- **EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION**
  In the method onDataObjectSendFragmentation, the fragmentation manager reads the data object content and generates a single fragment, after proactively fragmenting the data object if it was not fragmented before. The number of fragments depends on the files size and the fragment size. Different from network-coded blocks, fragments can be smaller than the (maximum) configured fragment size. A randomly selected fragment from the set of missing fragments according to latest Bloom filter information is then added to the event queue by means of **EVENT_TYPE_DATAOBJECT_SEND**. If there is no missing fragment left to send, an **EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL** is raised immediately. This is very different from network coding, where it is always possible to generate a new block.
- **EVENT_TYPE_DATAOBJECT_NEW**
The method `onDataObjectForReceived` is called if the `EVENT_TYPE_DATAOBJECT_NEW` event is raised (after reception of a data object and verification by the security and data managers) so that the fragmentation manager can gather all the fragments for a data object. Once it receives all necessary fragments they are reassembled to create a data object. If the original data object is reconstructed, the event `EVENT_TYPE_DATAOBJECT_RECEIVED` is generated (which will trigger verification of the reconstructed content by the security and data managers).

- **EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION_SUCCESSFUL, EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL**
  The method `onDataObjectSendSuccessful` intercepts the `EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL` event generated by protocol manager. It checks if the content object has been fragmented. If so it retrieves the corresponding original content object and raises `EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION_SUCCESSFUL` for processing by the forwarding manager.

- **EVENT_TYPE_DATAOBJECT_SEND_FAILURE**
  In a way similar to the success case, this event, if raised for a fragment, will be passed through to the forwarding manager as a `EVENT_TYPE_DATAOBJECT_SEND_FAILURE` event for the original data object.

- **EVENT_TYPE_DATAOBJECT_DELETE_ASSOCIATED_FRAGMENTS**
  This event is raised after a fragmented data object is successfully reconstructed. All associated fragments are removed and deleted from the fragmentation decoder storage.

- **EVENT_TYPE_DATAOBJECT_DELETED**
  Deletes the data object specified in the event from the fragmentation encoder and decoder storage. This event is raised by the data store, for example, if another component decides to delete the data object.

- **EVENT_TYPE_DATAOBJECT_AGING_FRAGMENTATION**
  This is a periodic timer that ages off fragments in the fragmentation storage layer which are older than a configurable maximum age parameter.

**New Classes**

The classes for fragmentation are organized and named similar to network coding, but for sake of completeness we give their brief descriptions here. The only differences are in the encoder and decoder services, which use similar names but the coding and encoding reduces to the special case of fragmentation and reassembly, respectively.

**FragmentationManager**: Maintains suitable instances of `FragmentationEncoderService` and `FragmentationDecoderService` on a per-data-object basis. It listens for fragmentation related events as described above. A manager module is created for the fragmentation encoder and fragmentation decoder in order to have a separate thread dedicated to both processing tasks.

**FragmentationEncoderService**: Generates a fragment from the given data object. Fragmentation of a data object is done proactively upon first invocation, but like
network-coded blocks, fragments are not inserted into the data base at the sender. Generates EVENT_TYPE_DATAOBJECT_SEND once the fragmented block is generated.

**FragmentationDecoderService**: Gathers fragments until all the required fragments are received to reconstruct the data object. Generates EVENT_TYPE_DATAOBJECT_RECEIVED for the generated data object.

**Modified Classes**

**ProtocolManager**: Added if statement to check if EVENT_TYPE_DATAOBJECT_SEND should generate EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION. Control message and node descriptions are never selected for fragmentation. Selective/adaptive and more general content-based application of fragmentation could also be implemented here.

**ForwardingManager**: EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION_SUCCESSFUL is treated as EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL, but the data manager is not inserting the parent data object into the Bloom filter of the peer, because it signals only partial success. We don’t know yet that the parent data object has been fully reconstructed at the receiver.

**Event**: Existing code has if statements checking for EVENT_TYPE_DATAOBJECT_SEND and perform work such as adding the node to the node list. The treatment of EVENT_TYPE_DATAOBJECT_SEND_FRAGMENTATION has been added to the same if statements.

**Protocol**: A performance optimization where the receiver uses a variant of Haggle’s REJECT message (called REJECT2) if the original data object has been reconstructed (which can be detected by a Bloom filter check) has been implemented here. This is short-circuiting the normal propagation of the Bloom filter to the sender for the one-hop case. The sender will raise a EVENT_TYPE_DATAOBJECT_SEND_SUCCESSFUL for the last fragment in this case, which the fragmentation manager translates into the same event for the original data object, which then is inserted into the peer’s Bloom filter by the data manager.
3.6.2 Sample Configuration and Parameters

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

```xml
<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 different from network coding not all fragments have to be of equal size. The last fragment of a data objects may be smaller that the specified fragment size.
3.7 Security

The confidentiality and integrity of content are particularly crucial in peer-to-peer contexts where some participants may not be trustworthy. The CBMEN ENCODERS security solution provides substantial value over the traditional approach, automating the process of ensuring that information is accessible to authorized principals and not to others. Using attribute-based encryption allows us to cryptographically implement complex security policies. Applications can dynamically set access control policies that are specialized according to the content and context. In addition to providing access to those that are authorized and denying access to all others, the figures of merit for our security approach are low latency and computational overhead when publishing and retrieving information.

3.7.1 Security Approach

This description is focused on key aspects of CBMEN ENCODERS Security, namely:

1. Integrity,
2. Trust Establishment, and
3. Confidentiality

**Integrity and Non-Repudiation.** Haggle creates a signing and verification key pair per node. When content leaves a node, its hash is signed. The input to the hash includes all the descriptive attributes as well as the content itself. Signature verification is a check that the content came from the node in the system that it is claimed to be from. Note that applications are nodes in Haggle, so this framework can provide end-to-end assurance. In principle, the granularity of authentication could be refined to be application-specific. Since node descriptions are sent frequently, they can optionally be sent without signatures to reduce load, using the below entry in the node’s configuration:

```xml
<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:

```xml
<SecurityManager signature_chaining="true" />
```
With this functionality, each node can re-sign the data object before forwarding it, adding its signature to a list. A subscriber only has to directly trust the last sender, and can decide whether to implicitly transitively trust the chain of signers along the path from the publisher, or to explicitly examine the list to make a choice. With this feature active, data object’s metadata will contain a chain similar to the one below:

```
<SignatureChain hops="2">
    <Signature hop="0" signee="..."> .... </Signature>
    <Signature hop="1" signee="..."> .... </Signature>
</SignatureChain>
```

**Decentralized Certification.** In Haggle 0.4, the same fixed certificate authority public and private keys were provisioned at all nodes. This was used to sign the public key for each node and to generate a corresponding certificate. These certificates are exchanged in node descriptions during encounters between devices. Since the same certificate authority key pair was present on all nodes in Haggle 0.4, a node would accept the certificate of any other node and thereafter trust it. Accepting all certificates effectively means that the signing mechanism did not provide any security guarantees. This was addressed by introducing a certification framework, where authorities each possess distinct Certificate Authority (CA) keys that are used to certify the public keys of other nodes.

For this purpose, we distinguish between nodes that are *authorities* versus those that are *users*. User A can get its public key signed by multiple authorities (and hence have multiple certificates). User B will trust user A if it trusts any of the authorities that have signed a certificate for A.

Any node can act as an authority, by setting a configuration parameter:

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

It will listen to requests for certificate signatures and respond to them appropriately. To ensure their confidentiality and integrity, 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. This is specified in a node’s configuration file with:
We provide the option for a node to boot in open certification mode (OCM). In OCM, a node will accept any certificates given to it. This is needed to initially bootstrap trust. The set of nodes that it will interact is limited (and controlled by physical proximity). It will also periodically broadcast its public key, waiting for authorities to sign it and return a certificate. This functionality can be activated in the configuration with:

```xml
<SecurityManager shared_secret="aGVsbG8gd29ybGQgMTIzNA==" />
<SecurityManager>
    <OpenCertificationMode enabled="true"
        certificate_signing_request_delay="60" first_request_delay="15" />
</SecurityManager>
```

It will implicitly trust any authority that returns a signed certificate (which is acceptable due to the use of shared secrets). Once sufficient signed certificates have been gathered, the node can reboot in closed certification mode, in which it will only accept node certificates that have been signed by at least one authority in its trusted authority set. Cached certificates will be ignored if no attestation from a trusted CA is present.

**Confidentiality.** The Haggle 0.4 code does not protect the confidentiality of content. However, access control must not become a single point of failure or limit network scalability. This precludes centralized solutions such as the use of traditional asymmetric ciphers that require the retrieval of the recipient's public key. While remote directory lookups can be avoided through the use of identity-based encryption (Franklin, 2003) that allows a sender to generate the recipient's public key locally, policy-based data sharing among a group would still require an alternate solution to eliminate the use of a trusted server (for group management and policy enforcement).

Our approach is to utilize a variant of ciphertext-policy attribute-based encryption (John Bethencourt, 2007), which lets a system-wide authority give each user decryption keys with appropriate attributes embedded in them. When content is encrypted, the sender defines the access policy in terms of the attributes that are needed to decrypt the content. The policy is a tree with operators (such as conjunction, disjunction, threshold, and range) at internal nodes and attributes at the leaves, allowing very expressive authorization. Since the policy is embedded in the content encryption process, enforcement is purely cryptographic and provides an end-to-end guarantee with no intermediate service needed to mediate access.

Cryptographic keys are typically certified hierarchically, with trust flowing statically from a single authority. In the past, a coordination phase has been required to enable interoperation between groups from different trust domains. **Multi-authority attribute-**
Based encryption (MA-ABE) (Allison Lewko, 2011) allows content to be encrypted with a policy framed over attributes from different authorities without any prior communication. We use MA-ABE to construct cryptographic capabilities for content, allowing trust to be managed orthogonally from key certification. This allows trust to be reconfigured locally, when the network is partitioned, by coordinated adjustment of the set of acceptable attributes.

In open authorization mode (OAM), 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:

```xml
<SecurityManager security_level="LOW" encrypt_file_payload="true"
  OpenAuthorizationMode enabled="true" attribute_request_delay="30" />
```

Access policies can be any Boolean formula over the attributes. Specifically, authority and attribute identifiers must be alphanumeric. The authority whose namespace an attribute is defined in must always be specified, with the two separated by a period. Thus, an attribute `Attr` from authority `Auth` is specified by `Auth.Attr` and is an atom in a policy. Attributes can be connected with OR and AND operators. Parentheses can be used to indicate precedence. The following is an example of a valid policy string:

```
Authority1.Attribute1 OR
  (Authority2.Attribute1 AND (Authority3.Attribute2 OR Authority3.Attribute9))
```

### 3.7.2 Using Access-Control Policies

Limiting access based on attributes provides a flexible framework. Traditional access control, where permissions are granted to specific subject-object combinations, can be implemented by using an access policy for the data object that requires the receiver have a key with the subject attribute. The use of groups provides syntactic sugar to ease the definition of access policies. Group-based access control can be implemented by encrypting the data object with an access policy that requires the receiver have the appropriate group attribute. Similarly, role-based access control (RBAC) can be implemented by requiring the recipient have the correct role attribute.

The policy is added as an attribute of the data object being published. The extended Haggle security manager uses the specified access policy to encrypt a key (used to symmetrically encrypt the content) with MA-ABE. The data object is routed from the publisher to the subscriber based on the recipient's interests. If the receiving node has the appropriate cryptographic attributes, it can decrypt the content.
3.7.3 Security Implementation and Haggle Integration

Background Signing. The default Haggle SecurityManager class signs data objects eagerly when they arrive from an application. It registers for EVENT_TYPE_DATAOBJECT_INCOMING, which indicates that the metadata of the data object has been received. The signature is computed over the data object's attributes and identifier, both of which are available in the metadata. This signature suffices as a tight binding to the content since the data object identifier is derived from the hash of the content and the attributes. In addition, the SecurityManager signs data objects that are about to be sent to a remote node if they have not already been signed. This is needed for data objects, such as node descriptions, that are generated internally within Haggle. To do this, the SecurityManager registers for EVENT_TYPE_DATAOBJECT_SEND, checks if the data object has been signed, and otherwise adds a signature and the node's certificate.

To sign a data object, the SecurityManager's onIncomingDataObject() and onSendDataObject() methods directly invoke the signDataObject() method in the Haggle's SecurityHelper class. This blocks the main Haggle thread, which must wait for the computationally expensive step to complete. Hence, we have used Haggle's design pattern of asynchronous manager modules that perform background processing. This is based on task queues that handle all computationally-expensive functions of the security manager. Instead of calling signDataObject() directly (as happened in Haggle 0.4), a SecurityTask of type SECURITY_TASK_SIGN_DATAOBJECT is enqueued with the SecurityHelper's addTask() method. The SecurityHelper's doTask() method, which dequeues tasks and processes them, had an undefined SECURITY_TASK_SIGN_DATAOBJECT case. This was modified to invoke signDataObject(). Similarly, the SecurityManager's onSecurityTaskComplete() method, which sends kernel events signaling the completion of security tasks, had an undefined SECURITY_TASK_SIGN_DATAOBJECT case. It was modified to send a kernel event of type EVENT_TYPE_DATAOBJECT_SEND after the signing task completes.

The ProtocolManager class has been modified to only handle event EVENT_TYPE_DATAOBJECT_SEND when the data object has been signed or is a control message. In either case control will pass control to the onSendDataObjectActual() method of the class. This will ensure that data objects destined for remote nodes are only processed by a protocol after they are signed.

Decentralized Certification. In version 0.4 of Haggle, every node was provisioned with the same CA keys, precluding meaningful certification. This has been addressed by allowing any node to act as a CA (by setting an authority name in the configuration file and generating distinct CA credentials). It can then serve as an authority for both certification and issuing MA-ABE attributes.

The certification procedure proceeds as follows. First, out of band, authority A and node N agree on a symmetric encryption key K1 and a hashed message authentication code.
(HMAC) key K2 (the implementation currently uses the same key for both). The node N will periodically broadcast a data object (with the `SecurityDataRequest` attribute) that contains its public key and node identifier. This object is encrypted with K1 and HMACed with K2. Any authorities listening to this request can take the public key, sign it, and publish a data object (with the `SecurityDataResponse` attribute) that contains the signed public key, again encrypted with K1 and MACed with K2.

During normal operation, when a node N1 and a node N2 first meet, they exchange node descriptions, that will contain all their signed certificates. When N1 receives certificates from N2, it will go through all the certificates, checking whether any of them are signed by authorities it trusts. If so, it will add N2 to the list of trusted nodes and save the public key. Further exchanges of data objects can proceed with N1 verifying that objects from N2 are signed with this public key.

With the availability of multiple certificate authorities, it is no longer possible for a node to verify a certificate signature until it has received the appropriate authority's CA certificate. Without having certificates stored in the certificate store, it cannot verify data object signatures. Thus, when a node joins a network, and is at a high enough security level, it will not accept node descriptions or data objects since it cannot verify the signatures on the data objects.

To handle this, a node can boot in open certification mode, where it will accept all certificates without trying to verify the signatures. It is assumed that this will be done with the user's consent when they join a new network so that Haggle can rapidly gather certificates for nodes. Once the appropriate certificates have been collected, the node can reboot without this option enabled. Subsequently it will only use node certificates that are signed by known authorities. Note that an authority will accept public key signature requests without checking for a signature on the request data object since it checks the HMAC with the node-specific symmetric key.

Lazy Encryption. Whereas signing a data object can be done after the metadata is received, encryption requires the availability of the content as well. This is signaled by `EVENT_TYPE_DATAOBJECT_RECEIVED`, which the SecurityManager already registers for. It then invokes `onReceivedDataObject()`, which can be modified to trigger encryption eagerly as soon as content from an application is available to the Haggle kernel. However, this would result in encryption of content that may never leave the node.

Instead, content can be encrypted lazily when it is being sent to a remote node. If an application sets the `Access` attribute, the SecurityManager's `onSendDataObject()` method will create a SecurityTask of new type `SECURITY_TASK_GENERATE_CAPABILITY`, and enqueue the task for processing with the SecurityHelper's `addTask()`. A new case for `SECURITY_TASK_GENERATE_CAPABILITY` has been added to the SecurityHelper's `doTask()` method, to invoke a new SecurityHelper method `generateCapability()` that will

(i) look for the capability corresponding to the Access policy in the cache; if found, the capability and extracted symmetric key will be used; otherwise a
new symmetric key is generated and MA-ABE used to encrypt the symmetric key with the Access policy,

(ii) store the resulting capability as an attribute of the data object.

(iii) enqueue a new SecurityTask of type SECURITY_TASK_ENCRYPT_DATAOBJECT with the symmetric key from step (i)

A new case for SECURITY_TASK_ENCRYPT_DATAOBJECT has been added to the SecurityHelper’s doTask() method, which calls a new SecurityHelper method encryptDataObject(). This encrypts the file in the DataObject with the symmetric key.

The ProtocolManager class has been modified to only handle event EVENT_TYPE_DATAOBJECT_SEND when the data object has been encrypted, in the case that the Access attribute is present and the data object is destined for a remote node. This will ensure that data objects are only processed by a protocol after the encryption completes (if it is to be applied).

If the SECURITY_TASK_ENCRYPT_DATAOBJECT task replaced the plaintext in the file specified by data object’s filepath field, local Haggle applications would no longer be able to access the content (through the filesystem). Simultaneously, if the same data object has to be sent again to a remote node, its content should not have to be encrypted again. This duality is handled by adding a new encryptedFilepath field to data object, and storing the encrypted data in a file specified in the new field. Prior to performing the encryption, a check is done to see if the encrypted version of the file is already present, in which case the encryption does not have to be performed.

The content of a data object is extracted from the filesystem for transmission to a remote node in the sendDataObjectNow() method of the Protocol class by calling the retrieve() method of the DataObjectDataRetrieverImplementation class. The constructor of DataObjectDataRetrieverImplementation has been modified so that it opens the file specified in the encryptedFilePath field if the data object is destined for a remote node and it has had encryption applied to it.

**Cached Decryption.** When a node receives encrypted content, it does not need to decrypt it (if it only needs to forward the encrypted data). Decryption is only needed when the data object is of interest to a local application. To check whether content is encrypted, the data object’s attributes are checked for the presence of a decryption capability.

The onSendDataObject() method of the SecurityManager class has been extended to check whether the data object’s metadata includes a capability and is intended for a local application, in which case a SecurityTask of new type SECURITY_TASK_USE_CAPABILITY is created and enqueued for processing with the SecurityHelper’s addTask(). A new case for SECURITY_TASK_USE_CAPABILITY was added to the SecurityHelper’s doTask() method, to invoke the new SecurityHelper method useCapability() that:
(i) extracts the access capability from the data object’s attributes,
(ii) checks the cache for the symmetric key corresponding to the capability; if not found, MA-ABE is used to decrypt the capability with the node’s attribute keys, and the symmetric key is added to the local cache,
(iii) enqueues a new SecurityTask of type SECURITY_TASK_DECRYPT_DATAOBJECT with the symmetric key from step (ii)

A new case for SECURITY_TASK_DECRYPT_DATAOBJECT has been added to the SecurityHelper’s doTask() method, which calls a new SecurityHelper method encryptDataObject(). This decrypts the file in the DataObject with the symmetric key.

Content for which the signature verification fails will always be discarded at the earliest possible time, but content for which the decryption fails should not be discarded. This is because the data object may need to be forwarded to other nodes that have the appropriate access rights, and the current node may obtain the required access rights in the future.

**Key Distribution** The SecurityDataRequest / SecurityDataResponse mechanism is used for key distribution. There are two types of requests, ones for specific keys or ones asking for all that are available. When an outgoing data object needs to be encrypted and the appropriate encryption attributes are not present, a SecurityDataRequest data object is sent for the specific attributes that are needed. Similarly, when a data object needs to be decrypted and the appropriate decryption attributes are not present, a SecurityDataRequest data object will be sent for those specific attributes.

The above mechanism suffices for providing the required functionality. An auxiliary mechanism has also been implemented to improve performance. When a node interacts with a new authority (after receiving a certificate signature), it will send requests to that authority to get all its attributes. Receiving these early will eliminate the need to request the attributes during a subsequent operational phase.

It is possible to configure the location used to store the temporary files containing requests and responses, as shown below:

```xml
<SecurityManager temp_file_path="/tmp"/>
```

The maximum number of outstanding security data requests can be specified as well:

```xml
<SecurityManager max_outstanding_requests="3"/>
```
Signature Chaining. The SecurityManager's `onSendDataObject()` method has been modified to support appending signatures to data objects when they are being sent. If signature chaining is enabled, then before sending a data object, a check is performed. If the signature on the data object was created by the local node. If not, it creates a `SECURITY_TASK_SIGN_DATAOBJECT` task that replaces the signature on the data object. After that, an entry is added to the `SignatureChain` metadata of the data object. The entry contains the node's signature. The result is that the recipient can inspect the `SignatureChain` metadata and find a list of signatures of the data object identifier, each from a node along the path it traversed.

### 3.7.4 Sample Configurations and Parameters

```xml
<SecurityManager security_level="HIGH" signature_chaining="true"
    sign_node_descriptions="true"
    max_outstanding_requests="3" temp_file_path="/tmp"
    encrypt_file_payload="true"
    shared_secret="YWJkOTFsYm5qejaZbmdtYw==">

    <Authority name = "SomeAuthority" />
    <OpenCertificationMode enabled="true"
        certificate_signing_request_delay="15" first_request_delay="15"
        certificate_signing_request_retries="2" />
    <OpenAuthorizationMode enabled="true" attribute_request_delay="30" />
</SecurityManager>
```

- **security_level** – specifies whether no data objects should be signed (LOW), only node descriptions should be signed (MEDIUM), or all data objects should be signed (HIGH).

- **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.

- **temp_file_path** – specifies the directory where SecurityDataRequest data objects are created, encrypted, and decrypted.

- **encrypt_file_payload** – if true, content encryption is activated; otherwise, content is sent without encryption.

- **shared_secret** – is the Base64 encoding of the secret key shared between an authority and user node. SecurityDataRequests and SecurityDataResponses between the two nodes will be symmetrically encrypted and HMAC'ed with this key.
name – the identifier by which the node will be known as an authority.

certificate_signing_request_delay - 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 – the number of retries to send certificate signing request. This is useful for testing purposes, the default is unlimited.

first_request_delay – the amount of time in seconds before the first request is sent to an authority, requesting it to certify this node’s public key.

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

3.7.5 Limitations and Possible Future Directions

We have identified a number of avenues for future research and optimization of the security functionality. They are outlined below.

Signatures. Haggle uses the RSA digital signature scheme. When a data object is sent, it is signed. The signature and the corresponding verification key are added to the metadata. This allows the receiver to verify the signature without contacting a remote directory of public keys. To reduce the storage overhead of including the sender’s public key certificates, identity-based signatures could be used. The receiver would then use the sending node’s identity to generate the public key. The reduction in storage overhead occurs at the increased computational cost of constructing public keys at the receiving nodes.

Encryption. Attribute-based encryption relies on pairing-based cryptography, which is computationally expensive. As the number of attributes utilized in an access policy grows, the time to encrypt as well as the time to decrypt an access capability grows proportionately. To mitigate this, sub-policy memoization can be investigated. In particular, it may be possible to decompose an access policy into policies that provide monotonically stricter access. Narrower sub-policies can then be used to distribute capabilities, which can be combined to provide flexible access while reusing cryptographic computation.

Policies. Haggle applications can now scope access to the content that they are publishing by defining the access policy as an attribute of the data object. It may be possible to enrich the access policy automatically by allowing the application to add a limited description, such as a label for a particular access policy. When content is being published, a limited closure computation could be used to map the description to an enriched version. This richer version would then be used when defining the cryptographic access for the data object. Only nodes with the right attributes would be
able to access the encrypted content. Such a process would allow access policies to be semi-automatically created and cryptographically applied.

**Key Distribution.** In the current implementation, a user shares a single shared secret with an authority; and at any given time it can only have one shared secret. When it wants to encrypt (or decrypt) an object with a policy that has attributes from multiple authorities, it broadcasts a SecurityDataRequest encrypted with the shared secret. This will only go to one authority (as the others will have different shared secrets). Consequently a node must reboot once it gets its attributes from one authority, replacing its shared secret with that of the next authority. The process continues till attributes have been collected from all authorities.

The procedure could be streamlined letting a node’s configuration specify multiple shared secrets, one per authority it wishes to contact. An authority would store a corresponding list of shared secrets. When a node requires attributes, it would broadcast multiple requests, one per authority from which it requires attributes.

**Signature Chaining.** The current signature chaining protocol allows a recipient to verify the signatures present. However, missing signatures from nodes that were traversed cannot be detected. Instead a protocol could be adopted where each hop signs the previous signature and adds that to the end of the chain.
The development of the Phase 1 CBMEN software has been completed on schedule, which is depicted below. Interest-driven content-distribution, proactive content replication, lightweight dissemination of node-descriptions, and content-based caching have been integrated with each other and into the Haggle framework. A performance evaluation of interest-driven routing in comparison with epidemic dissemination and an improved version of Haggle’s adaptation of PROPHET to content-based networks has been conducted. Adaptive interest modeling has been integrated into Haggle to perform smart pre-fetching and initial performance results have been obtained. Adaptive interest modeling has been generalized to support Drexel’s rich metadata and a new mechanism to share interest models has been added. Network coding and fragmentation (as well as their combination) have been completed and integrated with the other components. The performance of network coding has been evaluated relative to a baseline with atomic transfer and traditional fragmentation approaches, and in particular with regard to energy consumption. Haggle’s data object signing mechanism has been replaced by SRI’s new multi-authority security solution, which enables decentralized certification and policy-based content access. The detailed results of our performance evaluation have been made available to the government as part of the TIM and PI briefings and can be found in our final Phase 1 report that will be delivered under a separate cover.

An integrated version of CBMEN has been evaluated by the MIT Lincoln Labs team in several integration experiments and in the field at Ft. AP Hill. A VIP demo took place at Ft. AP Hill on May 29 that successfully and convincingly demonstrated the capabilities of all our components in the field and in four separate sidebar demonstrations.

<table>
<thead>
<tr>
<th>Component</th>
<th>Phases</th>
<th>04/12</th>
<th>05/12</th>
<th>06/12</th>
<th>07/12</th>
<th>08/12</th>
<th>09/12</th>
<th>10/12</th>
<th>11/12</th>
<th>12/12</th>
<th>01/13</th>
<th>02/13</th>
<th>03/13</th>
<th>04/13</th>
<th>05/13</th>
<th>06/13</th>
<th>07/13</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEGA</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRI Interim Assessments</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SARC Interim Assessments</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Government Assessments and Demo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>SRI Team</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interest-Driven Content Distribution</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>21</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Content Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opportunistic Caching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>Proactive Replication and Caching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cooperative Caching with UHEF Optimizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptive Caching (utilizing Interest Models)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptive Interest Modeling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Network Coding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Network Coding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Improved Network Coding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Content Signing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>Policy-Based Access Control</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

An integrated version of CBMEN has been evaluated by the MIT Lincoln Labs team in several integration experiments and in the field at Ft. AP Hill. A VIP demo took place at Ft. AP Hill on May 29 that successfully and convincingly demonstrated the capabilities of all our components in the field and in four separate sidebar demonstrations.
5 Summary & Conclusion

This design document is provided as a contract deliverable A0002 to describe the Phase 1 CBMEN ENCODERS design. Our development proceeded closely following our initial plan. Thanks to the use of an exiting open source framework all members of the SRI team and more generally of the MSI’s team were able to proceed in parallel to meet the milestones for all intermediate assessments Inside the SRI team we use a git-based workflow to support maximally concurrent development activities. In addition to a master branch for Haggle that is used for general improvements and bug fixes, we maintain a separate feature branch for each component (currently we are maintaining semantics, direct, caching, coding, imodel, security, encryption, and udp-bcast branches). Furthermore, when a component of another performer is released that our code depends on, we include it as a separate feature branch in our own git repository, as we did for the drexel branch. All our components are written with integration in mind. In most cases, integration boils down to a simple git-merge. This turned out to be the case for the integration of direct and caching in a new integration branch direct-caching. It also worked in the case of direct and network coding, which was integrated by a simple merge in a new direct-coding branch. Only a few minor problems were detected and quickly fixed. The integration with fragmentation was similarly straightforward. Also the initial integration of the Drexel registrar manager with distribution in a drexel-direct branch went smoothly. The integration has been further improved by disseminating rich metadata proactively and optionally through broadcast instead of relying on default interest for such data at each node. The integration of signing with network coding and fragmentation required some additions to pass through signatures. The integration between network coding and fragmentation was possible with hardly any changes in the components, which was quite surprising for us. Furthermore, it was instructive to see that it was possible to add UDP-based protocols, even broadcasting for node descriptions, without changes to the routing architecture. The imodel-drexel integration required most work because the interest modeling had to be extended by additional elements and functions to exploit rich metadata and submit rich queries, respectively. In addition, we are maintaining a semantics branch, which cleans up the Haggle semantics to bring it closer to its mathematical specification. This branch took more effort to integrate and test because it affects various parts of the system, but in the end it flawlessly worked together with all other components. The most recent addition is the encryption branch, which extends the basic security branch by multi-authority certification and encryption. The attribute-based encryption primitives are implemented through a crypto-bridge (provided by SAIC) to the CHARM library. The integration of encryption with the other features (in particular fragmentation and network coding) was non-trivial, but thanks to the modular design was possible without major problems.

Regarding the Haggle framework, our experience is quite positive. We have identified and documented a couple of limitations and gotchas for our team members and other
performers, and found a number of bugs, which for most part were due to small oversights and easy to fix. We had very few issues during the development, and our developers were quickly able to master the complexity of the framework and the new components thanks to the existing documentation and frequent discussions inside the SRI team. It also helped that we have hosted other team members as interns or visitors at SRI. The use of the Linux version of Haggle for rapid prototyping and testing in CORE Linux containers has saved valuable development time, because testing on Android phones (which we are doing as well) is more time-consuming.

All this has allowed us to focus on the difficult problems, which are mostly concerned with the right architectural decisions and algorithm design to allow maximum flexibility (e.g. mix and match or different components or features and full parameterization through the configuration, which in the future has the potential to become more dynamic and adaptive).

After the code drop for the first intermediate assessment, we have also made modifications to the Haggle framework to significantly improve its performance, which enabled us to run 30-node scenarios on a single machine with CORE and Linux containers. The performance improvements are also needed for a sensible component-level performance evaluation for networks of this size. For component-level performance evaluation and testing we have designed a set of scripted scenarios for CORE/EMANE that have been further extended as more features have been added.

In parallel with the performance evaluation on CORE we have ramped up our testbed to 30 Android phones and replicated SAIC’s IE1 and IE2 test cases. We have also conducted a smaller number of over-the-air tests to better understand the limitations of the CORE/EMANE models, in particular in regard to channel contention and packet losses. In addition, we have been running our own scripted tests using a small test application (called “haggletest”) that exercises various API features and is useful to stress the system to find subtle timing or multi-threading problems that do not show up with Linux containers on CORE.

In the second half of Phase 1, our test framework has been further extended to run automated parameter space exploration studies and regression tests based on concise scenario parameter specifications. The framework is based on CORE/EMANE with Linux containers and enables us to run a large number of tests on a set of Linux servers, typically overnight.

Preliminary performance results have been reported in the previous TIM and PI meetings. Our final Phase 1 report summarizes our performance results and includes detailed performance reports and papers in the appendix.
6 Bibliography


