Scalability Aspects of Agent-based Naming Services

Karl Kleinmann, Todd Wright, BBN Technologies, Cambridge, MA
{kkleinmann, twright}@bbn.com

Abstract—In this paper, we discuss the scalability of the White Pages (WP) service implementation of the Cougaar distributed agent architecture. We give an overview of its design and derive performance equations that provide a model-based estimate for both WP message counts and related message size in bytes. We present experimental results performed on two different agent societies of different scale to validate the performance equations. The experiments confirm the accuracy of the model and demonstrate the scalability of the design of the Cougaar WP naming service.

1 INTRODUCTION

Over the last decade, the field of scalable multi-agent systems (MAS) has come unto its own. Once the research community developed an initial understanding of issues involved in multi-agent cooperation and coordination, difficult research questions involving the scalability of different agent control strategies came to the fore. Results in scalability have been a perennial topic for conference papers since at least Agents 2000, and at the same time funding agencies started expressing explicit interest in scalability: DARPA’s ANT, ALP and UltraLog agent research programs have had the explicit goal of building military-grade agent frameworks that scale to hundreds or thousands of coordinating agents.

One of the recurrent themes in the scalability work to date is that techniques and tools that are successful at building cooperating agent societies of a few 10’s of agents are often problematic or infeasible when the number of agents (and the complexity of their interaction) rises by an order of magnitude or more. For simplicity, most agent systems employ basic centralized implementations for their foundational capabilities, such as directory services, QoS monitoring, matchmaking and brokerage. Scaling these capabilities for MAS, while complex, often involves the application of known software engineering techniques. However, engineering large-scale agent systems also involves prominent issues of resource allocation, fault tolerance, multilevel security, agent lifecycle administration, and monitoring tools. More interestingly, the realistic scaling properties of control mechanisms, messaging ontologies, negotiation algorithms, and other higher-level agent capabilities remain largely unexplored.

In this paper, we will discuss the scaling properties of an agent-based implementation of WP naming services. Naming services can be divided into Broadcast-based Discovery, Yellow Pages that support a more general directory service including attribute-based search, and White Pages [2]. WP are used to resolve symbolic names, such as agent names, to physical addresses, such as the protocol-specific host/port address where the agent currently resides [7]. WP are required in distributed applications where the addresses are either not known at startup or are dynamic at runtime. The most notable WP is the Internet directory service Domain Name System (DNS) [4], whose primary task is to resolve host names to IP addresses.

Section 2 will present the agent-based WP design that was developed using the Cougaar framework. Cougaar ([9], [11]) is an agent architecture for building large-scale MAS that has been sponsored by DARPA through its former ALP [15] and UltraLog [11] programs. Cougaar was designed to support data intensive, inherently distributed applications, where application scalability is paramount. Cougaar is available as Open Source and enjoys a worldwide user community. For a better understanding of the scalability issues, we briefly discuss the evolution of the WP design over four major design revisions in the last six years.

In section 3, we will develop performance equations that model the WP bind and lookup traffic. These equations, based on an analysis of the implemented WP algorithms, can predict the number of WP messages and the related message size in bytes. The parameters of the equations are specific factors defined by the application, constants due to the implementation, and configuration variables that can be chosen to tune the WP configuration in order to optimize system performance.

Section 4 will present experimental measurements obtained from runs with two Cougaar agent configurations of different problem sizes. A comparison with the numbers obtained by the equations proves the usefulness of the model as a predictor of WP traffic cost, and demonstrates that the WP implementation scales very well.

Section 5 summarizes the paper including the current design status of the Cougaar WP implementation, and gives an outlook on what we hope to accomplish with future research.
2 WHITE PAGES ARCHITECTURE

In Cougaar, the WP resolves agent names to Message Transport Service (MTS) addresses, such as RMI stubs, plus agent version information. Tables 1 and 2 give an example of the lookup result for a particular agent:

Table 1 - Lookup result for agent “DRT-HQ.ABV.MIL”

<table>
<thead>
<tr>
<th>Agent Name</th>
<th>DRT-HQ.ABV.MIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>UID</td>
<td>ABV-A-NODE/1101927845261</td>
</tr>
<tr>
<td>Data</td>
<td>&lt;Table 2&gt;</td>
</tr>
</tbody>
</table>

The WP is used by agent-level services, such as the MTS, and is implemented within Cougaar using agents. The advantage of an agent-based approach is that it allows the WP agents to leverage the capabilities of the underlying agent infrastructure, such as Cougaar’s advanced security services [10], [14].

The Cougaar WP design details are beyond the scope of this paper, but are well documented in [10], [12]. In this section, we cover the primary WP design features, with a focus on features that directly impact scalability and performance.

In Cougaar, a node is a single Java Virtual Machine (JVM) instance that may contain and maintain multiple agents. In most cases, there is a 1:1 correspondence between the node and the hardware platform. Every Cougaar node includes a client-side WP resolver, which caches data found in WP servers. Each client-side resolver has three major tasks: to lookup data from WP servers, to manage entries bound by local agents, and to select which WP server to use. A WP server is an agent that can run on any node in the agent society. Multiple replicated WP servers are supported, which forward data to one another to maintain a consistent and complete view of all agent records.

Client-side leases are used for data returned in WP lookup requests. These lookup leases allow WP clients to safely cache data until the lease’s Time-To-Live (TTL) expiration time, at which time the client can either discard the data if it was not used or send a WP message to renew the data. The use of caching improves overall performance and scalability. The TTL controls how long data can be cached, which is limited by how long the application will accept stale data.

In Cougaar the lookup TTL is typically set to 4 minutes, to speed up the detection of restarted agents and the new address of mobile or restarted agents. In some situations, the Cougaar message transport can identify when the data is stale and force a cache flush, which bypasses the TTL.

Server-side leases are used for all records bound in the WP. Each WP client node is responsible for renewing the leases bound by all local agents before the lease expires. WP servers are free to remove agent entries from their naming service tables if their leases are not renewed before the bind TTL. If this bind TTL is set too high, the WP servers will continue to list unresponsive or agents until the TTL passes. Server-side leases also guarantee that temporary server errors, possibly due to server crashes, are fully resolved once the TTL period has passed. In Cougaar the bind TTL is typically set to 8 minutes.

Both lookup and bind messages are batched, to reduce message traffic. In our experimental configuration (cf. section 4), we set the batching intervals to 75% of the TTL boundaries.

The Cougaar WP design also supports replicated WP servers, which are used to improve overall WP scalability and robustness. In this paper, scalability is defined as the application’s ability to support increased problem size, and robustness is defined as the application’s ability to recover from failure [13]. Replicated WP servers improve scalability by allowing servers to load-balance the clients that they must serve. Replicated WP servers improve robustness by allowing clients to switch servers at runtime, which is used when servers become unresponsive or are dead.

Figure 1 – Overview of the WP architecture

A review of the evolution of the Cougaar WP design highlights the scalability and robustness features of the current design as compared to the prior alternatives. The current
WP design is essentially the fourth refactor of the Cougaar naming services, with the original implementation dating back 6 years to the original ALP project [15].

The original WP implementation featured no caching or leases, and was not robust to server failures. This early naming service was only used by the early ALP project, which effectively cached protocol addresses from the naming service until an agent's protocol link stopped working. Naming service messages used an out-of-band RMI channel instead of the Cougaar message transport. This design scaled to 30 agents distributed across 5 hosts, but quickly ran into scalability problems once the naming service started being used by clients other than the message transport.

The second WP design added simple client-side caches with server-side change callbacks to "push" new data to the nodes. This design had a high startup messaging cost, since initially there are many WP changes, but at steady state the WP messaging overhead was very small. The primary downside of this design is that it taxed the servers to update clients.

The third WP design was a complete redesign to support client-side caching with leases and to use the Cougaar message transport for naming service messaging instead of RMI. This implementation is equivalent to the current WP design, but was limited to a single WP server.

The current WP design added WP server replication and client-side server selection. The scalability and robustness properties of this design are the focus of this paper.

### 3 Scalability Analysis

Based on knowledge of the basic design of the Cougaar WP, a “back of the envelope” set of performance equations were developed and are discussed in this section. The performance equations estimate the overall WP message count per second and byte per second, and therefore help analyze the scalability of the design.

Scalability factors, such as the number of agents and WP servers, are some of the basic variables of the WP performance equations. The primary benefit of these equations is that they predict the WP overhead and guide WP tuning options, such as the configuration of lease durations. These equations have been validated in our experiments, as we will discuss in section 4.

WP traffic can be divided into two types: bind (registration) traffic, to bind new names and renew bound leases, and lookup traffic, to resolve names and refresh cached resolver results. These two traffic patterns are discussed separately, since they have different communication patterns and messaging overhead. The cumulative traffic is simply the sum of these two WP traffic quantities.

The following table defines the variables used in the performance equations, and gives some typical values from our experimental results (cf. section 4).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
<th>Typical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Number of Agents</td>
<td>1146</td>
</tr>
<tr>
<td>N</td>
<td>Number of JVM Nodes</td>
<td>52</td>
</tr>
<tr>
<td>S</td>
<td>Number of WP Servers</td>
<td>7</td>
</tr>
<tr>
<td>X</td>
<td>For each node, % of total entries cached</td>
<td>10%</td>
</tr>
<tr>
<td>C</td>
<td>Cache lease duration (client side)</td>
<td>4 min</td>
</tr>
<tr>
<td>L</td>
<td>Bind lease duration (server side)</td>
<td>8 min</td>
</tr>
<tr>
<td>M</td>
<td>Memory for agent entry in WP message</td>
<td>78 bytes</td>
</tr>
<tr>
<td>H</td>
<td>Header size of WP message</td>
<td>640 bytes</td>
</tr>
</tbody>
</table>

#### Estimated Bind Traffic

WP registration traffic is required to bind agents into the WP tables and replicate these bindings to the WP servers. The WP design allows an agent to bind in any WP server, and requires WP servers to replicate bindings with their peer WP servers. The basic sequence is:

- once every bind lease renewal period \( L \):
  - for all \( N \) nodes:
    - (node sends 1 msg. with its agent's leases to a server) +
    - (server sends 1 acknowledgement back to node) +
    - (server forwards the leases to its \( S-1 \) peer servers)

The number of messages for bind traffic is therefore:

\[
\text{NumOfBindMsgs} = \frac{(S + 1) \cdot N}{L}
\]

The number of bytes for bind traffic is:

\[
\text{BindTrafficBytes} = \text{NumOfBindMsgs} \cdot (\text{HeaderSize} + \text{(AgentsPerNode} \cdot \text{BytesPerLease})) = \frac{(S+1) \cdot N \cdot (H + (A / N) \cdot M)}{L}
\]

Note that in our implementation the node’s message to the server is the same size as the server’s acknowledgement message that is sent back to the node. Both messages specify the agent names and the unique identifiers (UID), which are used to uniquely identify the record’s modification version. This server to node message could be optimized to refer to the node’s message, which would avoid the agent names and UIDs in the acknowledgement message. However, this would complicate the node, as it would need to maintain more state.

From the above equation, it is clear that WP registration traffic is proportional to the number of agents and servers, and inversely proportional to the lease duration. To tune the configuration, a user can increase the lease duration to reduce WP traffic, if stale data is acceptable, as noted in the prior architecture section’s design overview.
Estimated Lookup traffic

The basic sequence for the WP lookup mechanism is:

once every cache lease renewal period C:
  for all N nodes:
    (node sends 1 renewal msg. for all cache entries to a server) +
    (server sends 1 msg. back with data)

The number of messages for lookup traffic is therefore:

\[ \text{NumOfLookupMsgs} = (1/C) \cdot N \cdot (1 + 1) = 2 \cdot N / C \]

If we assume that a node keeps X percent of all A agents in its cache, the number of bytes for the lookup traffic yields to:

\[ \text{LookupTrafficBytes} = \text{NumOfLookupMsgs} \cdot (\text{HeaderSize} + (\text{CacheEntriesPerNode} \cdot \text{BytesPerLease})) = 2 \cdot N / C \cdot (H + ((X \cdot A) \cdot M)) \]

As in the bind equations, the lookup equations show that the WP traffic is proportional to the number of nodes, since there are per-node caches, and inversely proportional to the caching duration. The X \cdot A factor is critical, since if every node requires a full WP list then the performance is proportional to N \cdot A, which is large in practice.

The client’s choice of a WP server is also a significant factor. The above equations assume that all client-to-server network links are equivalent, whereas in practice these links may fall across significant bottlenecks, such as LAN/WAN boundaries. In our deployed configuration, the client-side server selection code sends “pings” to servers to estimate the round-trip-time, plus we further trim the list of candidate WP servers to remove known cross-LAN bottlenecks.

Scalability Implications

The bind and lookup equations show that the overall WP traffic is both proportional to application-specific factors, configuration variables, and implementation constants.

- A and X are application specific factors that represent the problem size.
- S and N are configuration variables that are selected in the society configuration. These variables typically grow with the problem size, which means that the WP scalability is not just linear in the number of agents. The tradeoff in S and N is to balance between robustness and load (large S and N) vs. reduced traffic (small S and N).
- L and C are also configuration variables, which can be used to reduce the WP traffic if necessary. The tradeoff in L and C is to balance between “freshness” of the cached data (small L and C) vs. reduced traffic (large L and C).
- M and H are implementation constants that are not affected by the problem size.

4 EXPERIMENTAL RESULTS

In this section, we present experimental results that validate the performance equations. We ran two Cougaar agent configurations of different problem sizes and measured the WP message traffic.

In order to apply the performance equations, we determined the implementation constants M and H by measuring the WP traffic in the larger configuration. As shown in figure 2, the constant values were measured as M = 78 bytes and H = 640 bytes.

![Figure 2 – Measurements of M and H constants](image)

Medium-Size Agent Configuration

The smaller configuration featured 115 agents distributed across 10 nodes with 3 WP server agents. The agent society was running a logistics application ([11], [3]), showcasing a military logistics planning and plan-execution system. This society was the prototype application of the DARPA UltraLog program, in which Cougaar infrastructure was developed to improve the survivability of MAS, including scalability, security, and robustness [8], [6].

The configuration was profiled to measure the ratio of the agent names cached in each node relative to the total number of agent names in the society. This ratio reflects the communication patterns of individual agents, and is a property of the application itself. In the medium-size agent configuration the average ratio aggregated at each node was measured to be X=25%.

Table 4 shows the predicted WP traffic according to the performance equation from section 3. Figure 3 shows the measured total WP traffic (bind + lookup), which was sampled at 4 times during the 90 minutes run.
There is an initial spike in both message count and size per message. The initial increased message count is due to the lack of batching for first-time WP lookups and bindings. The extra bytes per message are due to the requirement to send the full record data (cf. Table 2) in response to a first-time lookup or bind request, as opposed to the UID-only renewals used in later stages. Once the caches are initialized, the message counts and sizes are reduced and become constant.

If we compare the measured results (steady state) with the model-based prediction (cf. Table 4), it shows that the equations provide a reasonable estimate of both the number of WP messages and the total WP traffic in bytes. We believe that the differences are due to the simplifying assumptions of the performance equations, such as the assumption that the agents are equally balanced between the nodes, as seen in the bind equation’s A/N term, and the slight difference between batching intervals and lease TTL intervals.

Large 1000 Agent Configuration Experimental Results
For our second configuration, we ran 1146 agents distributed across 52 nodes with 7 WP server agents. As in the smaller application, the UltraLog logistics application was used, but configured to run a larger problem with more agents. The per-node X cache namespace ratio of all names was measured to be 10% on average. This is smaller than the 25% of the medium-size configuration, because the agents in the larger configuration are more specialized and better distributed.

Table 5 shows the predicted WP traffic according to the performance equation from section 3. Figure 4 shows the measured total WP traffic (bind + lookup), which was sampled at 4 times during the 90 minutes run.

Table 5 - Model vs. Measured Results for the Large-Size Agent Configuration

<table>
<thead>
<tr>
<th>Experimental Conditions:</th>
<th>A = 1146 Agents, N = 52 Nodes, S = 7 Servers, X = 0.1</th>
<th>L = 8 min, C = 4 min, M = 78 Bytes, H = 640 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results per Model (Prediction):</td>
<td>Bind Traffic 78 msg/min 372 kB/min</td>
<td>Lookup Traffic 26 msg/min 249 kB/min</td>
</tr>
</tbody>
</table>

As in the medium-size configuration, there is an initial peak in both message count and size per message, but after ~25 minutes the WP traffic reaches a steady state.

The comparison between measured results and prediction (cf. Table 5) shows that the performance equations provide a very good estimate of the real traffic. The forecast is better than in the case of the medium-size society due to the statistical effects of bigger numbers: deviations from the average and unaccounted factors do not impact the result as much as in the smaller society.

While the large society contains 10 times more agents than the medium-size society, the WP traffic in bytes increased by a factor of 14 (the model predicted factor 17). This confirms that the WP traffic is not just linear in the number of agents, but that for practical purposes, the implementation scales very well.
According to the performance analysis in section 3, these results (measured 14 / predicted 17) are a consequence of our configuration choices: the number of WP servers S increased from 3 to 7 and the number of nodes N increased from 10 to 52. Using less servers and nodes would reduce the WP traffic, but would put more load on each single instance, creating bottlenecks and degrading robustness. The chosen configuration presents a balanced tradeoff between the load on each host vs. the overall WP traffic cost.

5 CONCLUSIONS
In this paper, we discussed the scalability of the Cougaar White Pages (WP). We gave an overview of its design, including how the design evolved to support greater scalability. From an analysis of the implemented WP algorithms, we derived performance equations that provide a model-based estimate for both WP message counts and related message size in bytes. The performance equations can be used as a guideline to predict the WP overhead of a Cougaar agent society and to tune the WP configuration in order to optimize performance.

We performed experiments on two different agent societies of different scale to validate the performance equations. The experiments confirmed the accuracy of the model and demonstrated the scalability of the design of the Cougaar WP.

Future work includes support for WP namespace partitioning, most likely in the form of WP server hierarchies. We also plan to explore other replication techniques, such as distributed hashing [5], to reduce the dependence upon hierarchical names to balance server load. A partitioned namespace will improve the scalability of the design by distributing load, but will add overhead in smaller configurations that could run with the current non-hierarchical replicated WP design. The hierarchical WP design will closely mirror the naming hierarchy design of DNS, including the use of dot-separated agent names. To prepare for this enhancement, the current UltraLog application already uses hierarchical agent names.

6 ACKNOWLEDGEMENTS
This work was sponsored, in part, by the DARPA UltraLog contract #MDA972-01-C-0025.

7 REFERENCES