A Framework for Multiresolutional Knowledge Dissemination and Collection in Dynamic Agent Societies

Edward Benson, Jeffrey Berliner, and Marshall Brinn
BBN Technologies, Cambridge, MA
{ebenson, berliner, mbrinn}@bbn.com

Abstract—*The dynamic, distributed, and autonomous nature of agent-based applications make knowledge extraction and distributed processing a complicated task. This paper presents Fido, a programming model and framework implementation that simplifies this task by concealing the details of the agent society under a fixed processing pipeline. Fido developers are then free to focus on the business logic of the application with very little overhead and additionally benefit from multiresolutional result formation, distributed result caching, and other features of the overall framework.*

1. INTRODUCTION

Agent architectures present a powerful means to design and construct distributed applications, but their dynamic, distributed, and autonomous nature create a number of problems when interacting with the system or subsets of the system. As a result, existing distributed processing frameworks generally require some centralized foreknowledge of the component topology, such as service capability or data location, to handle complex question or task. This centralized control violates the fundamental agent paradigm and limits a society's inherent scalability, parallelism, and dynamism.

This paper presents the Fido framework, a package of plugins for the Cognitive Agent Architecture (Cougaar) [1] that addresses the problems of dissemination and collection from a local, agent-level perspective. Fido was created out of the need to aggregate various types of information across a large, distributed agent-based logistics application. Agents in this particular application were allowed to form dynamic command relationships, stored and maintained locally, so no assumptions about agent topology could be known by the user posing the query.

Rather than address the specific problem that faced this particular application, we designed and implemented a framework that attempts to address some of the challenges in agent computing that made this task so complex. Fido is an asynchronous request-response framework that conceals the particulars of a Cougaar agent society by allowing users to pick any one agent as an entry-point into the society and deal only with that agent. By using a small set of classes that allow tailored business logic over a fixed distributed processing pipeline, users can form complex distributed workflows without ever dealing with the details of the agent society itself.

The remainder of this section discusses relevant background information, followed by an introduction to the Cognitive Agent Architecture. Section 2 provides a detailed look at Fido's design and benefits. Section 3 describes Fido's plugin architecture, and Section 4 concludes with examples of Fido's real world use and future areas of improvement.

Background

Grid-based approaches to distributed processing, such as the Globus Toolkit, commonly separate and define grid members based on their technical role within the grid [2]. The Open Grid Services Architecture defines notions such as service factories, registries, and hosting environments that create an environment in which batch processing is a commodity easily scheduled and moved from machine to machine [3]. In these environments, locating resources is a well-defined task, as service discovery services and data indexing services are available to transform a workflow specification into a set of resource endpoints [4] [5].

Google's MapReduce [6] framework is a particularly interesting grid-like approach to data processing. Rather than offering service directories of pre-made or commercial processing tools, MapReduce provides a fixed pipeline into which users must plug in their own task implementations described in a variety of supported languages. By imposing rigid restrictions upon the way in which workflows can be described, MapReduce is able to hide the fact that it operates within a distributed environment, allowing developers with little distributed programming background to write processing tasks that run in a large distributed environment.

Cougaar

The Cognitive Agent Architecture (Cougaar) is a scalable, distributed, and open source, multi-agent architecture developed over an eight-year period by BBN Technologies [7]. Cougaar supports agent environments ranging from embedded applications to globe-spanning agent societies.

The complete network of Cougaar agents in a distributed application is called a *society*. Each agent maintains relationships with other agents, belongs to groups of other agents with similar interest known as *communities*, and is hosted on a physical *node*. Using a flexible configuration scheme, developers construct their distributed application on top of the society as a set of *components* that provide the agents, communities, and nodes with interests, capabilities, and behaviors [8].

Plugins are special components that are added at the agent-level to contribute a piece of business logic. Plugins have access to a number of services hosted on each agent that provide capabilities such as society-wide time synchronization and generation of unique object IDs. The Blackboard Service serves as the agent's persistence mechanism and provides publish-subscribe capabilities that activate plugins based on interests it has registered with that service. Interest is registered using a Unary Predicate pattern that matches against objects that have been added, deleted, or modified on the Blackboard [8], [9].

Cougaar agents are autonomous, but are able to communicate over a variety of mediums using Cougaar's Message Transport Service (MTS). Agents can also form dynamic role-based relationships that often come in the form of a producer-consumer or superior-subordinate pair.

2. FIDO DESIGN

Fido is a framework for intelligently distributing small pieces of code for distributed execution and reporting. In the spirit of Google's MapReduce framework [6], Fido is intended to provide developers with a great deal of flexibility by simplifying the problem of distributed agent processing down to a single predictable pattern of operations. In Fido, this pattern is: *Request*, *Assess, Expand, Combine, Report*.

When a Fido-enabled agent receives a Fido *request*, it first *assesses* the request to determine if it can answer it immediately using cached results or its own capabilities. If not, it dynamically generates a list of agents needed to help answer the request and then *expands* the request into sub-requests destined for each agent in that list. Upon receiving the partial results from each agent, it *combines* them into an aggregated result, performing work and possibly adding its own piece as well. Finally, it *reports* the full or partial result by replying to its requestor.

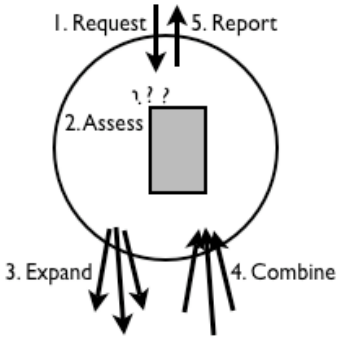

Figure 1 - Fido Interaction Sequence

Users interact with this pipeline by implementing or configuring objects that control the decisions made at each step. From the developer's perspective, using Fido only requires the direct use of four objects: the *Request*, the *Expander*, the *Worker*, and the *Result*. Requests initiate Fido work, maintain process configuration, and carry the Expander and Worker as payload. Expanders assess the request in the context of its owning agent and generate a list of other agents, if any, that should be incorporated into the workflow. Workers contain the code that collects data, performs work, and/or distributes information on each agent. Finally, Results contain the output (if any) of the Fido work that was performed and contain the business logic that specifies how to combine themselves with incoming partial results. Fido's underlying framework takes care of orchestrating and executing all of the interactions involved. Figure 2 depicts the model from Figure 1 from the perspective of object interaction.

To permit maximum flexibility within this fixed pipeline, all objects that implement a Fido workflow are described in regular Java code rather than a query or business process language. The Java-based approach allows Expanders and Workers full access to the Cougaar infrastructure, and it has the additional benefit of allowing Fido to be not only a query framework, but also a framework for distributed processing and society-wide information dissemination.

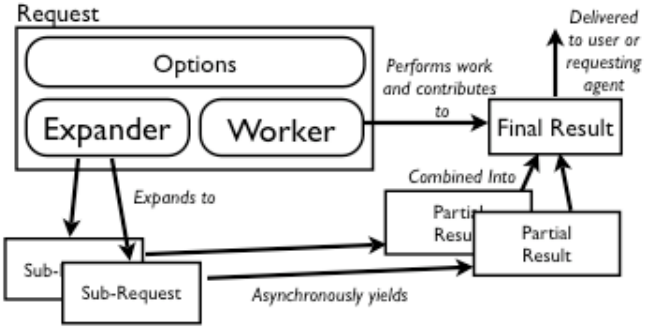

Figure 2 - Core Fido Objects

The Request object comes packaged with the framework, and in practice a project might re-use two or three

application-specific Expanders. The effort required to use Fido then largely consists of implementing a Worker and its corresponding Result object. Once these two interfaces are implemented, users can disseminate and query information across an agent society with no knowledge of its topology.

Fido's object design and workflow process afford it several attractive attributes, including:

Parallel, Asynchronous, Distributed Workflow—As Fido Requests are expanded, agents perform work in parallel and respond to their requests asynchronously. Fido's infrastructure further allows agents to participate in multiple workflows simultaneously and, if necessary, to participate many times within the same workflow.

Multiresolutional Results—The manner in which Fido results are assembled leave a trail of partial results chained across the agent society. Each agent that becomes a part of a Fido workflow generates its own partial result. In addition, each expanding agent maintains a partially aggregated result that incorporates all partial results "downstream" of that agent.

The result reporting and storage mechanism is therefore both distributed and multiresolutional. In agent societies dispersed across physically disparate locations, this presents a result storage process that is survivable in adverse environments. This redundancy helps to ensure that *some* result is available under all conditions, even if that result is not the most complete possible.

This design allows users to "drill down" into details of a result for more information as needed. A Fido-based result in an interstate highway monitoring society might provide final results on a per-state basis. Seeing a problem in Virginia, a user could drill down into the Virginia partial result by visiting the appropriate agent, without the need to re-query the society. The Virginia-level result would show that the source of the problem status was coming from Northern Virginia. Further drilldown into the region-specific agent would show actual traffic backup data from I-495.

The stored chain of aggregated and partial results throughout the society thus provides several ways to look at the output of a Fido workflow. Different portions of the results leading up to the final result vary both in their completeness and in their scope, so all are potentially valuable pieces of information. Figure 3 depicts the quality of result improving as combination occurs throughout a society.

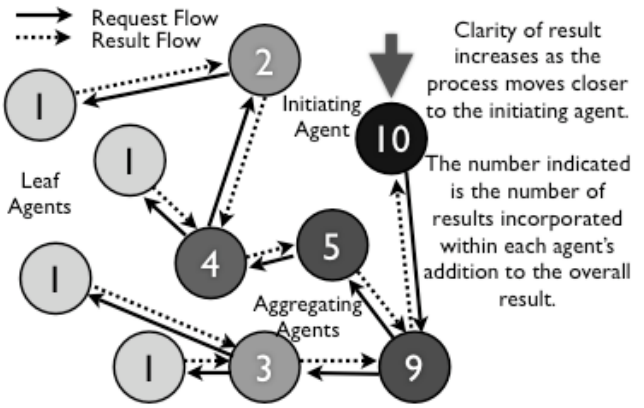

Figure 3 - Fido Result Flow

Topology-Independent Execution—Because expansion decisions are made locally on each agent during the execution of a Fido workflow, the user initiating the request requires no knowledge of the particular agents and relationships involved in answering the request. Choosing between which Expander implementation to use or writing a new Expander is the closest users come to making binding and scheduling decisions.

Dynamic, Adaptable Workflows—A fourth benefit to the organic manner in which request expansion unfolds is its ability to respond to societal changes in real-time. Expansion decisions are made based off of the real time relationships, capabilities, load, and availability (among other things) of agents known to the expanding agent. If these characteristics change after a Fido workflow has been initiated, the Expander will simply use the most current information available.

The remainder of this section will provide more detail about use and operation of the four key objects from a user's perspective of Fido. Section 3, Fido Architecture, will then discuss the plugin architecture that manages Fido processes.

Request

Fido Request objects are used to initiate any work done by the framework. Fido Requests contain a Worker instance to perform the distributed work, an Expander to route the process throughout the society, and a number of settings that modify the behavior of the framework's handling of the request.

Some of the most important options included in the Fido request relate to caching and partial result handling. Fido caches past results based on a request fingerprinting mechanism within the framework. If a new request matches the fingerprint stored on a cached result and that result reports that it is not stale, Fido can return the cached result and avoid a possibly long and involved work process. Options on the request object control whether cached results

can be used, how thoroughly any newly generated results are cached, and how cached results are cleaned up. Overlapping with the cache features, the Request object also provides various options that dictate the way in which multi-resolutional partial results are maintained throughout the society.

Result

Result objects are the output of the Worker and represent either a particular agent's results from the work process or the combination of several agents' results. Workers and Results come in pairs, each Worker implementation is expected to have a corresponding Result implementation that it knows how to generate.

The Result interface only contains two requirements that the user must implement (other maintance-related requirements are handled by an abstract base class). A Result must know how to incorporate a child partial result of the same type with the `combine(Result result)` method, and it must be able to evaluate itself to determine whether or not its contents are stale for the purpose of caching.

While not all Fido tasks are result-producing queries, Result objects are not optional. Even if the work being distributed does not produce any output, some lightweight empty result must be implemented. In these cases it is common to use the Result as a mechanism for logging the names of the agents on which the Worker process was executed.

Results are combined at each junction of Expansion within the workflow. Along a particular path of Expansion, result combination occurs as the outermost Expander objects determine no more expansion is necessary. Partial results then follow the reverse path of expansion, combining along the way for increasingly more complete aggregated results.

Result Types

Fido contains five classifications of Results, reflecting the two ways in which a Result may be produced and the two ways in which a Result contributes to the overall Fido task, in addition to a special case. Results that require no expansion are called Elemental Results. These results can be provided immediately upon request during the *Assess* step of the pipeline by invoking the Worker object. Results that require the incorporation of input from other agents are called Aggregated Results.

The decision of whether a Result must be Elemental or Aggregated is not intrinsic to a particular request, result, or expansion, alone, but is a product of the condition of the society at the time of execution. The same request on the same agent might yield an Elemental Result or an Aggregated Result at different times due to changes in agent relationships, capabilities, knowledge, and topology.

A special case of Result is the "Contributing Result." This is the individual contribution of an agent at a point of expansion within the workflow. When the agent collects and combines expansion results, it adds its Contributing Result into the Aggregated Result. Table 1 summarizes the various result configurations.

Table 1 - Result Types

	Partial	**Final**
Elemental	Partial-Elemental	Final-Elemental
Aggregated	Partial-Aggregated	Final-Aggregated

Special Case: Contributing

Results are deemed Partial or Final based on their contribution to the overall Fido Task. Only one Final Result exists per Fido Request; all other results output by the task are Partial Results. The way in which each of these five results is preserved during and after a Fido workflow is controlled by options set on the Request object.

Expander

The Expander is the key design component that enables localized decision-making in Fido's request dissemination process. Its job is to determine which targets, if any, are the next steps in the unfolding work process.

Expansion is dynamic, so events that occur after the start of a Fido process can alter the manner in which problem-solving unfolds. As workflow is expanded throughout the agent society, the problem-solving process forms an organically grown directional graph outward from the initiating agent until the outermost Expanders independently determine no further expansion is necessary. As each path of expansion reaches an end, the process reverses direction as each agent aggregates the responses it received, makes its own response contribution, and responds to its requester.

The Expander interface requires just two variations of the method `expand(...)` for two different modes of execution that Fido supports. This stateless method accepts the agent's Service Broker, which provides access to any of the agent's local services (such as the Blackboard Service or Alarm Service), and it returns a list of agent addresses representing any Expansion that is required before a result can be generated. This output is one of the key deciding factors in the *Assess* step of the pipeline that determines if a Request can be handled immediately or requires more work.

Fido makes no restrictions on the manner in which tasks

expand. This leaves the possibilities of Expander implementation entirely open to the developer but introduces the possibility of infinite cycles within workflow expansion. While cycles in a controlled manner are not necessarily bad, future versions of Fido will likely contain some limit on the number of times they are allowed to occur (we have not yet found this necessary within our own Fido applications).

Worker

While the Expander, is responsible for determining the routing of Fido workflow throughout an agent society, the Worker is responsible for performing the actual work to be done at each agent. Like the Expander interface, the Worker interface only contains one method with two signature variations: `work(...)`. Both variations accept the agent's Service Broker to provide access to agent-level services and return a Result object.

Workers represent the business logic of a particular Fido task. They execute within a Blackboard transaction, so they can publish, modify, or gather information without worrying about other agent services concurrently modifying the agent's state. Much of the learning curve for Fido users is learning how to express a problem in such a way that it can be enclosed inside a Worker object. Since the same Worker is executed on all agents that are part of a Fido workflow, the `work(…)` method must be able to use the agent's services to determine when it is appropriate to execute or gather particular pieces of information and how to formulate that information into a Result object.

An Example

Listing 1 contains a very simple example of a Worker, Result, and Expander implementation that could be used to fetch and aggregate fuel levels across a Fido-enabled society. The base classes that the Worker and Result extend provide several framework-related functions required by the interface but not unique to a user's particular implementation. The Expander is intended to imply a society in which agents maintain a hierarchy of administrative relationships with each other. Submitting a request with the Worker and Expander in Listing 1 would result in the aggregation of fuel levels across the subtree of the administrative hierarchy rooted in the initating agent.

```
class FuelLevelWorker implements Worker,
                      extends WorkerBase {
  public Result work(ServiceBroker sb) {
    double level = assessLevel(sb);
    return new FuelLevelResult(level);
  }
  // .. Methods to assess the fuel level
}

class FuelLevelResult implements Result,
                      extends ResultBase {
  private double _level = 0;
  public FuelLevelResult(double level) {
    _level = level;
  }
  public void combine(Result other) {
    level += ((FuelLevelResult)other)._level;
  }
}

class AdministrativeSubordinateExpander
      implements Expander {
 public Collection expand(ServiceBroker sb) {
   return fetchAdminSubordinates(sb);
 }
 // code to fetch administrative subordinates
}
```

Listing 1 - Representative Example

Listing 1 shows how little overhead is required to turn a business logic component into a distributed Fido process. Application-specific logic goes within the Worker and, to a lesser degree, the Result, and helper functions to direct the workflow within the society go within the Expander. The Expander, it should be noted, is where all of the underpinnings of Cougaar's architecture are most often exploited.

3. FIDO ARCHITECTURE

The underlying architecture that automates the processing of Fido workflows is composed of three agent-level Cougaar plugins. These plugins manage the handling of requests, the aggregation of results, and the notification of result completion. These plugins can be added to agents during configuration time or dynamically injected at run time.

The `RequestHandlerPlugin` subscribes to incoming Fido Requests on each agent's Blackboard. When a Request appears, the plugin uses the enclosed Expander, the request's fingerprint, and Fido's Blackboard cache to assess whether the request can be handled immediately. If so, the plugin executes the request's Worker or obtains a cached result and formulates an immediate response.

If the Expander yields a list of additional society targets or caching is disabled, the `RequestHandlerPlugin` spawns sub-requests off of the incoming request and uses Cougaar's MTS to send them to the appropriate targets. After expansion, the plugin places a record of its expansion on the Blackboard and retires, becoming quiescent until a new Request object wakes it again.

When an agent responds to a sub-request, the Result object is sent via MTS back to the Blackboard of the requesting agent. Its arrival triggers a Blackboard subscription that wakes the `ResultHandlerPlugin`. This plugin keeps track of incoming partial results and, when all partial results have returned, initiates the combination process.

The first step in combination is for the expanding agent to execute the Worker locally to perform its portion of the work and obtain its contribution to the result. After this execution, the `ResultHandlerPlugin` takes this contributing result and uses it as the base result into which incoming Partial Results are combined, iterating over the partial results and passing each one into the `combine(...)` method on the contributing result. Finally, it publishes its completed result to the Blackboard, finalizes the expansion record, and transmits the result to the requesting agent.

A final plugin, the `ResultNotifierPlugin`, handles the asynchronous notification of final result completion. This plugin subscribes to final results appearing on the agent Blackboard and notifies any system threads that are waiting on the request object associated with this final result. Code that relies on Fido can either simulate a synchronous query process by blocking on the request notification or can operate asynchronously by subscribing to the Blackboard for the result object. We have found the `ResultNotifierPlugin` particularly useful in allowing Fido to expose its services over a Servlet interface. The servlet issues a request to Fido and then blocks until the result arrives, allowing the asynchronous workflow to be wrapped in a synchronous HTTP operation.

4. Conclusion and Future Work

Fido has become a successful part of distributed agent applications at BBN, enabling developers to treat arbitrarily large, complex, and dynamic agent societies as if they are a single entity that can be operated upon directly. It has been built into large-scale, agent-based logistics simulations in which agents may be hosted at the physically disparate locations that they represent. These scenarios model logistics efforts, such as hurricane and tsunami relief, dividing the work among agents that represent real-world entities.

In such applications, Fido provides a mechanism for distributing high-level control operations, such as the signals related to logistics planning, and performs maintenance operations, such as flushing objects off of the Blackboard. It also serves as a mechanism for assembling dynamic visualizations of society status to allow users to evaluate the status of various echelons and entities within the simulation and iteratively narrow down on problem areas. Finally, it provides a simple aggregation framework through which users may ask questions such as, "how many penicillin shots are left in New Orleans?" Hidden from the Fido user in all of these cases is that the work required to achieve the desired result takes place across potentially hundreds of agents.

Limitations and Future Work

While Fido has proven to be a useful addition to the Cognitive Agent Architecture, there are many opportunities to improve upon current capabilities and expand upon the framework:

A Workflow-wide Communications Channel—While Fido's agent-level perspective is often a strength, some mechanism through which messages could be passed within an existing workflow could do much to enrich Fido's capabilities. Such a channel might allow for more intelligent expansion decisions that incorporate the status of other branches of the workflow for load balancing and efficiency purposes.

Subscription-Style Fido Processes—The current version of Fido is strictly a request-response mechanism. This has proved valuable in many application areas at BBN and has highlighted the need for continuously running subscription-style workflows. These processes would begin with a request-response pattern, but would continue to update the response as partial and contributing results changed.

Greater Context in Requests. Expanders can make expansion decisions based on an agent's full suite of services, but the current framework does not allow them to append data to the sub-requests that get sent to these agents. As a result, contributing agents are merely told that they are a part of the workflow, not *why* they are a part of the workflow. Giving the expander the ability to add this context to individual sub-requests would lead to a more capable framework overall.

References

[1] Cognitive Agent Architecture. http://www.cougaar.org.

[2] I. Foster, C. Kesselman. "Globus: A Metacomputing Infrastructure Toolkit," Intl J. Supercomputer Applications 11(2), 115-128, 1997.

[3] I. Foster, C. Kesselman, J. Nick, S. Tuecke. "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," 2002.

[4] M. D. Beynon, T. Kurc, Catalyurek, et al. "Distributed processing of very large datasets with DataCutter," Parallel Computing 27, 11, 1457-1478, October 2001.

[5] T. Bellwood, L. Clement, C. von Riegen, et al. UDDI Version 3.0.1: UDDI Spec Technical Committee Specification. Organization for the Advancements of Structured Information Standards, 2003.

[6] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters," Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04), 137-150, December 2004.

[7] A. Helsinger, M. Thome, T. Wright. "Cougaar: A Scalable, Distributed Multi-Agent Architecture," 2004 IEEE Conference on Systems, Man and Cybernetics (2), 1910-1917, October 2004.

[8] *Cougaar Architecture Document*. BBN Technologies. December 2004.

[9] *Cougaar Developer's Guide*. BBN Technologies. December 2004