Fletcher: A Web Service for Planning

Thomas M. Vernieri  
Department of Computer Science  
North Carolina State University  
tmvernie@ncsu.edu

R. Michael Young  
Department of Computer Science  
North Carolina State University  
young@csc.ncsu.edu

Abstract

Artificial intelligence planning has traditionally been realized in applications that are tightly bound to planning systems. In this paper, we describe Fletcher, a Web service that exposes and augments a planning system. Fletcher provides an alternative to including a planner implementation within applications. In addition to benefiting applications that need to execute planning tasks, Fletcher gives human authors the option to be involved in the planning process directly.

1. Introduction

Web services have been instrumental in allowing applications to enhance their functionality by composing openly available components. Artificial intelligence (AI) planning systems have the potential to be used in a range of conventional applications, but they are rarely used outside of specialized intelligent systems. The work we describe here makes planning more accessible to developers of conventional applications by providing access to a planning system through a Web service named Fletcher.

Using a Web service as an interface to a planning system is a significant shift from previous methods for including AI planning functionality in applications. The typical method for including planning functionality in an application is to compile the planner into the application at build-time. This restricts the application developer in a number of ways. First, it requires either that the application be built using the same programming language used to construct the planning system or that the application developer connect his or her code to the planning system via a cross-language interface. Second, the application must share computational resources with a planning system whose runtime requirements are usually quite demanding. Finally, even though several instances of an application may be performing similar or identical planning tasks, the planners are isolated from each other, potentially resulting in duplicated work. For example, if two colleagues are collaborating on a task by each using an application containing a traditional AI planner, the two application instances may both do the same computations at different times during the collaboration. Since the planning system is part of the application, there are no built in mechanisms to allow one instance of the application to take advantage of the results obtained by the other. Since planning tasks often consume a large amount of computational resources, the ability to share information across planning-enabled applications could be a substantial runtime benefit.

Applications that use Fletcher include in their code a lightweight Fletcher client responsible for communicating with the Fletcher Web service; in Section 5, we discuss third-party tools that automatically generate this client code for a variety of programming languages. This service-oriented approach has two main benefits over a design requiring that a planning system be built into an application. First, applications that include a Fletcher client can discover a Fletcher server at runtime, allowing them to select a server that can provide them with the lowest computational load, highest processor speed, or lowest usage rates. Second, Fletcher client applications can delegate the often-substantial computational cost of a planning system to a server rather than consuming computational resources on its own processor.

Fletcher also gives the developer substantial flexibility when it comes to controlling and gathering information about how the planning problem is solved. When solving a planning problem, Fletcher uses an underlying model of planning as a search through a space of plans. The messages that a client can send to Fletcher determine how the search space of the planning problem is explored and provide the client with detailed information about the portions of the search space the planner has computed.

Fletcher’s flexibility provides different clients with distinct methods for accessing its services. For example, an application may access Fletcher in a manner similar to the way that conventional planning systems are accessed—by requesting that Fletcher construct a solution plan for a planning problem. Additionally, applications may use more of Fletcher’s functionality to control the planning process to a fine degree, for instance, by requesting incremental planning efforts or by parceling out elements of a large planning problem to separate Fletcher services. Further, an application may expose the communication between server and client to its user via a graphical user interface and allow the user to provide
recommendations and constraints about the planning process. This approach would integrate well with existing approaches to advisable planning [8]. In Section 6, we describe two scenarios of how applications use Fletcher.

2. Related work

Fletcher builds on work related to plan-space planning, planning languages, and Web services as they relate to planners.

2.1. Plan-space planning

Fletcher uses a hierarchical causal link planner named Crossbow, a C# implementation of the Longbow planning system [14]; Crossbow uses refinement search [6] as a model of the planning process. Refinement search is a general characterization of the planning process as a search through a space of plans. A refinement planning algorithm represents the space of plans that it searches using a directed graph called the plan space graph; each node in the graph is a (possibly partial) plan. An arc from one node to the next indicates that the second node is a refinement of the first (that is, the plan associated with the second node is constructed by repairing some flaw present in the plan associated with the first node). In typical refinement search algorithms, the root node of the plan space is the empty plan containing just the initial state description and the list of goals that together specify the planning problem. Nodes in the interior of the graph correspond to partial plans. Leaf nodes in the graph correspond to complete plans (solutions to the planning problem) or plans that cannot be further refined due, for instance, to inconsistencies within the plans that the algorithm cannot resolve.

Crossbow builds its search graph using best-first search. A heuristic search function ranks plans on the fringe of the graph and the most promising node is selected from the fringe, and using the heuristic search function to rank all the plans associated with the nodes on the fringe of the plan space,

1) using the heuristic search function to rank all the plans associated with the nodes on the fringe of the plan space,

2) selecting the most promising node from the fringe, and

3) using the domain to create all of the plan’s refinements, and then adding those refinements to the plan space graph, creating a new planning context.

Planning is typically initiated by creating a planning context containing a graph with a single node whose plan has only two steps: an initial step encoding the current state of the problem and a terminal step encoding the goal state of the problem. Planning can also be initiated in Crossbow by seeding the plan associated with the initial node with additional steps and structures. This approach is useful, for instance, when a partial solution to a planning problem is already known.

2.2. Planning languages

The requests that Fletcher clients send to the Fletcher server and the responses that they receive in return contain data about the client’s planning problem. Many other planning systems represent their inputs and outputs using PDDL [7] or similar languages and formalisms. Unlike these approaches, we use schemas written in the XML Schema Definition Language [12] to describe what data is valid for Fletcher messages. Fletcher does not use PDDL or another established planning language because these approaches do not lend themselves to use in Web services. In contrast, XML Schemas are ideal for use with Web services. By using XML Schemas to describe the data that Fletcher sends and receives, we are able to include that schema information directly in the Web service description (a WSDL document described in Section 5). This allows client applications and the Fletcher server to use XML validators to check the integrity of their data. Initial work by Gough [4] is intended to provide an XML replacement for PDDL; this work is still in its early stages, but as XPDDL matures, we expect to extend Fletcher’s capabilities to support it as an additional input format.
2.3. Planning Web services

Several other research projects have provided Web-based access to a planning system. A recent implementation of the O-Plan planner provides a Web-based interface accessible through CGI scripts [11]. At the time the interface was designed, no clear Web services standards had been established. Using CGI scripts increased the planner’s accessibility, allowing clients to invoke it over the Web rather than as a local application. However, beyond the benefit of remote invocation, the use of CGI as a programmatic interface did not increase O-Plan’s accessibility for remote client applications. Unlike Web services that use WSDL to describe themselves, O-Plan specifications are provided through human-readable documentation. Developers of applications accessing services described in a WSDL document can use third-party tools to aid in the task of writing a client; developers of O-Plan client applications must build their interface to the planner by hand based on specifications provided by user manuals that describe the CGI interface.

Tsoumakas et al. describe HAP-WS, a Web service that wraps around the HAP planning system [13]. In an approach similar to the one we use in the design of Fletcher, HAP-WS provides a WSDL service description for the Web service, which receives and responds to messages using SOAP. Since HAP-WS uses these two standards, developers can make use of existing developer tools to generate client code and client applications can easily discover the service at runtime.

Unlike Fletcher, however, HAP-WS is a relatively simple service that accepts only one message: a request for a solution to a planning problem. This request is composed of 1) a domain definition, 2) a problem definition, and 3) parameters for the HAP planner. The service responds with a string that describes the solution to the planning problem, which was computed by an instance of the HAP planner running on a server. While there are scenarios where this amount of functionality is sufficient for applications, there are a number of use cases where applications could benefit from a planning Web service with a wider range of features. In the remainder of the paper, we describe Fletcher’s design and functionality and demonstrate how applications can benefit from its feature set.

3. Implementation

Interactions involving Fletcher follow a client/server model. The Fletcher Web service (or simply Fletcher) is the server half of this model. There are three industry standards that are central to Fletcher specifically and to XML Web services in general: XML Schema [12], the Simple Object Access Protocol (SOAP) [1], and the Web Services Definition Language (WSDL) [2]. We use XML Schemas to constrain Fletcher’s input and output data. Fletcher and Fletcher clients communicate over HTTP, sending and receiving SOAP messages. Fletcher uses the SOAP document-style encoding for its messages rather than an RPC (Remote Procedure Call) encoding in order to maintain interoperability with clients written in a variety of programming languages. All of the messages that Fletcher supports consist of a request from the client to the server, followed by a response from the server to the client. The WSDL document that describes Fletcher specifies the names and content of the messages that Fletcher can receive and the responses that it sends.

Fletcher servers associate most client interaction with a particular session. Sessions are created by a Fletcher server when it receives an initial message from a client. Both server and client use session cookies in the SOAP header to manage sessions and associate messages with them. By using sessions, clients can create planning contexts and then perform sequences of operations on those planning contexts; Fletcher maintains a session until the session is marked as inactive (that is, after no messages have been received containing that the session’s session cookie) for a period of twenty minutes.

Since sessions are not intended to be long-lived, Fletcher allows clients to save and restore their planning contexts to a database. Whenever a client sends a request to save or load a planning context, Fletcher requires that the request message’s SOAP header include authentication credentials. In its current implementation, Fletcher uses a simple username and password for authentication. Planning contexts are loaded from the database only in response to requests containing the same username and password as that which was contained in the request that initially saved it. In future versions, a more sophisticated authentication and authorization strategy will be adopted in order to a) provide better security and b) to allow clients to share their stored planning contexts with other authorized members of a work group.

When a client sends a request that Fletcher save its current planning context to the database, the response contains a key that uniquely identifies the stored planning context. At any point after Fletcher saves a planning context, any properly authenticated client can provide that key to load it from the database into the working memory of its current session.

4. Messages

Fletcher accepts five different types of messages, each requesting a different operation related to the planning process. As described in Section 2, planning contexts are the basic objects upon which Fletcher and its underlying
planning system operate. Clients can send messages to Fletcher requesting that it

- create planning contexts
- respond with information about a planning context
- expand the explored search space of a planning context
- save and load planning contexts to and from a database
- provide a list of the names that identify the heuristic search functions available to the planning system

Bundled together, the first three classes of messages compose the functionality of a typical planning system. It is common for a planning system to provide a single function (or at least, a small number of functions) to access this functionality. Fletcher accepts a number of different message types that accomplish those tasks in different ways, giving the user relatively specific control over the tasks’ performance.

The following subsections further explain the classes of messages that Fletcher accepts.

4.1. Creating a planning context

The first step in using Fletcher to compute a solution to a planning problem is for a client to establish a planning context. One message type used to establish a context contains a domain, a problem, and the name of a heuristic search function. Fletcher uses the operators in the domain to create new plans during the refinement search process. Fletcher uses the problem to construct an empty plan with appropriate initial and goal states. As described in Section 2, this empty plan forms the root node of the plan space associated with the planning context. The specified heuristic is used to guide the planner’s search through the plan space of this context. The heuristic name that the client provides must match one of the heuristics that exist on the server hosting Fletcher (a message for retrieving the valid heuristic search function names from a Fletcher service is described in Section 4.5).

When Fletcher receives a message, it uses the corresponding XML schemas to check that the input is valid. If it is not valid, then Fletcher responds with a SOAP fault, explaining why the request was not processed. If the input data is valid, Fletcher creates a new planning context and sends the client a response that contains an identifier to be used to refer to the new planning context. By using the session cookie and the planning context identifier in future requests, the client can work with that planning context for the duration of its session. Clients can create multiple planning contexts by sending more than one of these requests; for each request, Fletcher returns a new planning context identifier.

A second message type used to establish a planning context includes slightly different arguments. In this message type, a client sends a domain, a partial plan specification, and the name of a heuristic search function. Fletcher establishes a planning context as in the prior message type’s case, but rather than use a problem to create the root node of a plan space, the partial plan specification is used to create a root node already containing plan structure. This message type is used to achieve the “seeding” approach to refinement search discussed in Section 2.

Fletcher provides a number of variations on these two basic messages; they serve as a convenience for the client and help to minimize the amount of data that must be transmitted between the client and the server. One option is for clients to send XML that encodes the domain information as well as either the problem or partial plan information; this XML must conform to the corresponding schemas. Alternatively, clients may first compress the XML using gzip and send the Base64 encoding of the resulting binary data, reducing the size of the transmitted data (usually resulting in a message that is one tenth of the uncompressed size). Finally, clients may provide planning context identifiers referring to planning contexts created by the same client earlier in the session. Fletcher will copy the indicated information from those existing planning contexts; by specifying a combination of new information and references to information already established in other planning contexts, a client is able to send less data when creating planning contexts.

4.2. Retrieving planning context information

At any point after creating a planning context, the client may request information about the context’s state. Fletcher accepts requests both for information about the context’s plan space as a whole and for information about a particular plan node in the space. When requesting information about an entire plan space, the client only specifies the planning context identifier in the request. Fletcher responds to this request with an XML document describing the tree of plan nodes that the planner has constructed (see Section 4.3). Each plan node that this document describes is identified by a plan node ID value. The plan space description also contains a ranking value for each plan node. This ranking value is computed by the planning context’s heuristic search function. When requesting information about an individual plan node in the space, the client specifies both the planning context identifier and the plan node ID in the request. Fletcher responds to this request with an XML document describing the plan represented by that plan node.
4.3. Exploring a search space

After creating a planning context, the client can instruct Fletcher to explore the search space in order to find a solution to the context’s planning problem. Two types of messages exist to initiate this activity, both of which have more than one variation.

One type of message is a request to refine a particular node in the context’s search space. The client must provide a plan node ID to identify the node to be refined. As a convenience, Fletcher also accepts a message indicating that it should refine the plan node on the fringe of the plan space that is currently ranked as the best plan by the context’s heuristic search function. Fletcher responds to the client with a message containing the ID of the plan node that was refined.

The other type of message is a request to expand the explored portion of the search space. This type of operation repeatedly refines the best plan until a termination condition is met. The client can tell Fletcher to halt its search a) as soon as it finds a complete plan, b) once it has refined a certain number of nodes, or c) once a certain amount of processor time has transpired on the server. If the client requests that Fletcher halt as soon as it finds the first complete plan, then Fletcher will either return the ID of the plan node representing the complete plan that it finds or an error value if it cannot find a complete plan. This error condition occurs if Fletcher searches every node in the plan space, if its search exceeds the given time limit, or if its search exceeds a server-specific resource bound. If the client requests that Fletcher expand the search space for a certain number of refinements, then Fletcher will respond with the number of nodes that it actually refined, which may be less than the requested number if the time limit was reached or if the search space was exhausted.

4.4. Using the database

Since sessions are invalidated after twenty minutes of inactivity, applications may need to work with the same planning context across multiple sessions. Fletcher allows clients to save planning contexts to a database and to load them later. Save and load requests both require a message header that holds username and password parameters; Fletcher only allows a client to load a planning context if it supplies the same username and password as was given when that planning context was saved. In our current implementation, the Fletcher server must know the username and password before it can be used by a client. Fletcher’s current techniques of authentication and authorization are very simple; we plan to extend them in future versions.

The request to save a planning context contains the planning context identifier as input; the response from Fletcher is the ID of the new database entry. The request to load a planning context contains the ID of a database entry as input and the response from Fletcher contains the new planning context identifier that specifies the loaded planning context. Note that a planning context identifier is only meaningful during a single session while the ID of an entry in the database refers to the same entity irrespective of the session.

4.5. Retrieving heuristic search function information

When creating a planning context, clients must specify the name of the heuristic search function that should be used during the planning process. The name that the client specifies must match the string name of a search function that is implemented on the Fletcher server. Clients can send a message requesting a list of the available heuristic functions’ names and Fletcher will respond with an array of strings. The client can then use any of those strings when sending a request to specify the use of a heuristic search function when creating a new planning context. The client is not required to include session information with this request and the response will not include session information.

5. Implementing a Fletcher client

A Fletcher client sends requests and receives responses composed from the types described in Section 4 above. Implementing a Fletcher client is made easier because of Fletcher’s use of SOAP and WSDL. Since Fletcher uses these industry standards, developers can leverage existing development tools when implementing a Fletcher client. These tools generate code for C#, VB.NET, or Java based on a WSDL document. Some available tools are:

- Web Services Description Language Tool (a Microsoft tool for .NET),
- WSDL2Java (an Apache Axis tool for Java), and
- WSContractFirst (a thinktecture tool for .NET).

After using one of these tools to generate client code, application developers can write an application that sends messages to Fletcher and receives responses by simply calling methods on a local object. In the following section, we describe usage scenarios that detail exactly how an application may use the Fletcher Web service through this client code.
6. Example scenarios

When an application uses the Fletcher Web service, the sequence of messages that it sends will follow a common pattern. These two scenarios describe concrete examples of how applications use Fletcher.

6.1. Planning a cut scene

Suppose that a computer game must display a short cinematic sequence in the middle of its execution; in the gaming industry, this is referred to as a cut scene. The game employs this cut scene in order to achieve certain goals, namely that the player knows the location of objects and that non-player characters in the game have performed prescribed actions. The state of the game’s virtual world at the beginning of the cut scene may be different each time the game runs since the player may perform different actions each time he or she plays the game. Because of this variability, the details of the cut scene cannot be included in the game when it is designed. Instead, the game expresses its need for a cut scene as a planning problem and uses Fletcher to find a plan for the cut scene.

When the game has enough information to formulate the cut scene planning problem, it constructs a domain and a problem in XML that is valid according to Fletcher’s schemas. The game also supplies the name of a heuristic search function when requesting that Fletcher create a planning context. While it could query Fletcher for a list of available heuristic search functions at runtime, in this scenario we assume that the game developer has configured the game with the URL of a Fletcher service and the name of a valid heuristic at build-time.

With this information, the game constructs a SOAP message requesting the creation of a new planning context. The game uses an HTTP request to send the SOAP message to Fletcher. Fletcher processes the message and sends a new SOAP message in response. The game receives this response and extracts from it the planning context identifier and the session cookie. The game constructs a new SOAP message requesting that Fletcher expand the identified planning context until it finds a complete plan. The game includes the session cookie in the headers of this SOAP message, and then uses an HTTP request to send this message to Fletcher. Fletcher receives this message and expands the planning context’s search space until it finds a complete plan. After the planner has found a complete plan, Fletcher responds to the HTTP request with a SOAP message that includes the plan node ID of the plan node representing that complete plan. The game processes this response, extracting the plan node ID.

The game can now send a final SOAP message to Fletcher requesting the details of the complete plan. The message contains the plan node ID and its headers include the session cookie. When it receives Fletcher’s response, the game executes the steps in the plan node, resulting in a cut scene that satisfies the game’s original goals.

6.2. Authoring a domain

For some planning situations, it is difficult to construct the domain and problem. We are developing an application that will assist human authors in the task of authoring these components. This application gives its users the ability to inspect the plan space as Fletcher explores it. The user makes small changes to the domain and problem and the application instructs Fletcher to make a new planning context. The user can then control how Fletcher explores the search space; the application achieves this by sending Fletcher refine or expand messages followed by requests for plan space and plan node information. This iterative process allows authors to gain an understanding of the search space (and thus the solution plans) that will result from a domain and problem.

7. Future Work

In future work, we will further exploit Fletcher’s potential as a central server for planning-related tasks. Currently, multiple clients can connect to Fletcher simultaneously to work on different planning contexts. We are extending Fletcher to support multiple clients collaborating on a single planning context. This will allow, among other features, groups of users to collaborate on the construction and use of Fletcher plans. In addition, we will enhance the authentication and authorization system for saving and loading plans. With support for more complex permissions, Fletcher will allow a client to make a planning context available to a group of collaborators.

With these features, Fletcher will begin to build a large repository of fully and partially explored planning contexts. Such a repository will facilitate data mining to gain more insight into the practical matters of planning. It will also be useful for case-based planning approaches (e.g., [5]), with clients searching planning contexts to find planning problems similar to their own rather than requesting that Fletcher solve their problems from scratch.

Fletcher’s treatment of heuristics is another area that is a good candidate for enhancements. In the current implementation, Fletcher has access only to a set of functions pre-defined on the server. There are frequently cases when a client would like to provide a heuristic that is customized to its particular application’s needs. Fletcher could be enhanced by allowing the server to accept a full heuristic specification from the client,
perhaps by defining a specialized and appropriately restricted language for evaluating plan nodes.

8. Conclusions

We have presented Web services as a method for delivering planning services to applications. Including a Fletcher client in applications allows developers to benefit from AI planning without attaching a full planner implementation to their applications. This ease of use combined with the broad range of functionality that Fletcher provides makes planning more practical for application developers than it has been in the past. As a Web service for planning, Fletcher moves beyond previous work and has the potential for future enhancements; this merger of AI techniques with Web-centric technology brings an AI planning system into the world of Web services.

9. Acknowledgements

This work has been supported by National Science Foundation CAREER award 0092586 and by Microsoft Research's University Grants Program. Thanks go to James Niehaus and Jim Thomas for helpful discussion on the design of Fletcher and to David Burke and Brian Shiver for their work on Crossbow. Special thanks go to Justin Harris for his insight on the workings of Fletcher and Crossbow, and for his efforts developing and extending the Crossbow implementation.

10. References


