CS 342: Security and Programming Languages
Project Summary: The Reform Programming Language

May 3, 2007

Dustin Graves
dgraves@gwu.edu

Abstract

Reform is a domain specific programming language designed to monitor, manage, and manipulate data streams. Data extracted from an active stream may be reformatted and redirected to one or more different data streams through functionality incorporated by the language to facilitate the definition, creation, and manipulation of interfaces to data streams and data formats affiliated with data streams. The language core consists of an interpreter and a set of components providing basic stream management capability, which may be extended through modules that are dynamically loaded at runtime. Communication channels may be established between different interpreter instances allowing remote manipulation of an interpreter, with language provided authentication routines to restrict remote access to local resources.

1.0 The Reform Language

Reform is a distributed, interpreted language incorporating functionality for managing and manipulating data streams. Each active interpreter instance acts as an independent component of a larger distributed network of interpreters. An active interpreter instance, known as a Reformer, depends on three key components to provide data stream interactivity and inter-interpreter communication. The Source component provides data stream management, the Format component provides data interpretation, and the Channel component is a specialized Source that controls remote communication. An additional component provides authentication for remote language commands.

The Source component provides functionality to receive and transmit data through existing data streams. Sources may be created, destroyed, and linked. Creating a data source provides an interface for reading from and writing to data streams. Sources are assigned an ‘in’, ‘out’, or ‘inout’ mode at time of creation to indicate the direction of communication provided by the stream. Data received by a source is stored in a temporary cache, with adjustable size, to provide a dynamic snapshot of the stream state. Procedures for filtering, processing, and converting data may be attached to Sources to be applied to data before it is transmitted or after it is received. Linking sources together allows data received by one source to be redirected to another.

The Format component assigns meaning to the data received from a stream. Formats provide structure to the raw stream data, making it possible to interpret, process, and manipulate data that is extracted from a stream. Existing formats may be mapped to each other, allowing data to be translated from one format to another so that data from one stream can be used with other streams. Formats, which can be predefined or defined at runtime, are registered with Sources to facilitate the decoding and encoding of raw stream data. Characteristics of the raw stream data, such as fields indicating a data type, are used to determine the exact format to be used for data translation.

The Channel component provides a method for remote Reformer manipulation. Channels are specialized sources that provide persistent two-way communication between Reformers. Channels can be opened and closed. Language commands are transmitted through Channels to remote Reformers for processing. Commands transmitted to remote Reformers are subject to authentication. Authentication is performed at the keyword, source, format, and channel level. Reformers exchange credentials to be used for authentication when establishing channels.

2.0 System Design

The core Reform system design provides a set of interfaces and abstract components which define a general system for processing stream data. The core Source, Format, and Channel components can be extended to allow incorporation of proprietary components for communication, processing, and authentication. The language interpreter is defined as a Singleton providing an interface to core language features to be accessed by local and remote command processing units. Monitors asynchronously receive data from Sources which are constructed by Factories. A collection of implementations for common Source and Channel components is provided with the core language.

The Source interface specifies methods for sending and receiving data. Components for communicating through sockets, pipes, COM ports, and files are derived from the Source interface. The Channel interface is an extension of the source interface defining methods for persistent two-way communication. Channels operate as a client-server system requiring a Listener to accept connections from remote Reformers.

The Format template defines methods for mapping raw binary and ASCII stream data to a structured record. The format contains a collection of fields with an ID, type, and value to represent individual data elements. A unique ID field distinguishes between different objects contained by an individual stream. Procedures for decoding and encoding raw data, mapping data to other formats, and filtering data can be defined for each Format. Conditional logic and flow control incorporated by the language may be employed to simplify data translation.

Factories provide a generic system for creating Sources. Each Source must provide a Creator capable of constructing the Source from a variable length list of parameters. Creators may be registered with the Factory at any time to allow the dynamic loading of Sources at runtime. When Sources with ‘in’ and ‘inout’ modes are created, they are associated with a Monitor to asynchronously receive data from the stream. Monitors apply filters and perform type conversion before redirecting received data to the Sources linked with the recipient. Format conversion requirements are specified when sources are linked.

3.0 Implementation

A basic interpreter supporting a subset of the Reform language, including much of the core functionality, has been implemented for the C# platform. The interpreter consists of an execution manager, parser, and core module providing support for the management of data sources and channels. Full support for fundamental language concepts such as conditional logic and flow control is not yet incorporated by the current implementation.

The execution manager receives language commands, either locally through a console or remotely through a Channel, and dispatches them to the parser for processing. A delegate is provided by the execution manager to allow remote commands received by Channels, which belong to the core module, to be dispatched to the parser. The parser for the interpreter was created with the ANTLR Parser Generator to process language commands and execute the corresponding core module procedures, accessed through the Reformer Singleton. The core module contains the components responsible for managing the state of the Reformer.

The core module contains implementations of the Source, Format, and Channel interfaces with concrete implementations of file and UDP socket Sources and a TCP client-server Channel. A Generic Factory interface with a concrete implementation of a Source Factory provides capability for creating Source components and a Channel Listener implementation accepts connections from remote Reformers. Source and Channel monitor implementations provide the functionality for receiving, processing, and sending data. An Authenticator interface is defined and extended to implement a read-only Authenticator. The Reformer Singleton provides a common interface to the core module components through methods associated with the following key words.

- Sources are managed with the ‘create’ and ‘destroy’ keywords.
- Sources are linked with the ‘->’ operator.
- Listeners are managed with the ‘listen’ and ‘deafen’ keywords.
- Channels are managed with the ‘open’ and ‘close’ keywords.
- Channels, Sources, and Formats are listed with the ‘channels’, ‘sources’, and ‘formats’ keywords.

- Individual Channels, Sources, and Formats are accessed with the ‘channel[]’, ‘source[]’, and ‘format[]’ keywords/operators.
- The ‘print’ keyword will print descriptions for channels, sources, and formats.

Future work will focus on the implementation of full support for data reformatting, filtering, and processing. Support for fundamental language concepts providing conditional logic and flow control will be added to make Reform a more complete and well rounded programming language.