Reform Data Processing System

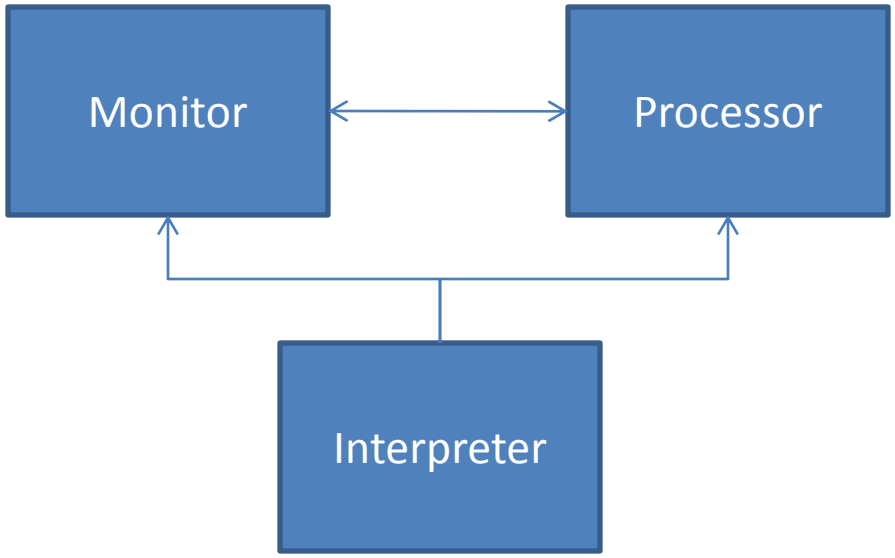
Dustin Graves

Design Overview

- The main Reform system is composed of three core modules
  - Language Interpreter
  - Data Stream/Source Monitor
  - Data Processor/Forwarder
- The Source Monitor and Data Processor form the underlying data processing component of the Reform system and are capable of operating independent of the Reform language
  - Can be treated as a framework to be used with other languages
  - Can be incorporated by applications requiring a generic system for data processing and reformatting
    - Exporting/Importing files to media players, word processors, etc

Design Overview

- The Source Monitor and Data Processor interact with each other, but do not directly interact with the Interpreter
- The Reform Interpreter controls the states of the Source Monitor and Data Processor
  - Provides application layer to the Reform system



Source Monitor

- The Source Monitor is responsible for monitoring the state of a data source and dispatching event notifications when an action is pending for a source:
  - Pending read – data is available for reading
  - Pending write – source is ready to accept data for writing
  - Pending connection – server socket has pending client connection to accept
  - Connection complete – non-blocking connection to server is complete
- Each source contains a “waitable” handle that can be monitored by the underlying operating system
  - File handles to be monitored by UNIX “select” system call
  - HANDLE object to be monitored by Windows “WaitForMultipleObjects” system call

Handling Events

- The Monitor can take two separate actions when handling a pending event
  - Read data from the source and send packet containing the received data, source ID, and event type to the Processor
    - Data writing must be handled separately as a special case
    - Delays due to event handling will interfere with source monitoring and possibly degrade performance
  - Queue source for processing by a separate thread or by the Processor
    - Variations for handling queuing sources that must be evaluated
      1. Monitoring is suspended for a source in the queue until the event has been processed
      2. Monitoring is not suspended for a source in the queue; the source only has one entry in the queue and a counter is employed to indicate any pending actions identified after the source was added to the queue
         - » Note that a unique queue entry consists of the pair of source and event type, not the source alone; source may have only one entry per event
      3. Monitoring is not suspended for a source in the queue; source can have multiple entries in the queue and is re-added to the queue when a new pending action is identified

Source Queuing: Variant One

- With variant one, two options are available to the data consumer
  - Consumer can read a single message from source, then re-register source with Monitor
  - Consumer can read all available/pending messages from source, then re-register with Monitor
    - This case is problematic as high rate sources that will always have pending data, such as files which have pending data until EOF, will create an infinite loop condition
    - This can be easily verified through testing with file I/O
  - The single read per event is the option to be used if implementing variant one
    - Option two will not be considered as an option for implementation

Source Queuing: Variants Two and Three

- Unlike variant one, variants two and three explicitly define the number of messages to process when handling a single event
  - Variant two reads *n* messages per event, where *n* is the number of messages pending for the source since its addition to the queue
    - Size of the queue is reduced with 1 entry for *n* events
    - Order of processing is altered at the source level, as precedence is given to sources near the front of the queue whose events are processed before sources near the end of queue, regardless of actual order of event occurrence
  - Variant three reads 1 message per event, but *n* events are present in the queue
    - Preserves the order of event arrival among sources
  - Use of variant three is preferable to variant two because it preserves the event ordering

Source Queuing: Selecting the Best Variant

- Although in theory variants two and three appear to provide higher throughput by never suspending monitoring of a source, in practice both variants suffer from a critical problem that excludes them as choices for implementation
  - Because some sources will continuously generate event notifications for the same event until it is processed, it is possible that *n* entries for the same event will be placed in the queue
  - Monitoring of a source must be suspended until the pending event is handled to ensure proper operation
- Variant one is the only viable choice for the Source Monitor implementation
  - Variant one will be implemented with the “read one message” per event mode

Variant One Comments

- The event handling variant which suspends monitoring of a source until the current event has been handled is the only viable choice for implementation with the Source Monitor
  - A single event will be processed for each notification
    - This appears to require that a single message be read from a source per event, but the programmer is provided with a workaround which provides some control over the number of messages read per event
    - When implementing the Source object, the ‘read’ and ‘write’ functions may be implemented such that multiple messages are received or sent at once
  - Research should be conducted to measure the performance characteristics of the Source Monitor
    - Number of sources and data rates that can be handled simultaneously
    - How multiprocessing can help to handle large workloads

Source Monitor Implementation

- The Source Monitor will be implemented with either the Reactor or Proactor design pattern, depending on the capability of the underlying operating system
  - Both patterns provide the same functionality with slightly different implementations
    - The Reactor pattern employs system calls to monitor the handle for a source, dispatching events to an event handler as they are encountered
    - The Proactor pattern employs asychronous source monitoring and processing capability provided by the operating system which will wait on an event, process the event, and forward the results to a consumer
  - The Proactor allows efficient use of available system capability to reduce the level of effort required by the Source Monitor
    - Due to implementation differences and lack of availability with some systems, he Proactor based implementation will be highly dependent on the underlying system
    - A Reactor implementation will be used for systems that do not support the Proactor model, and can be used as a generic solution on systems for which the Proactor implementation is pending

Source Monitor Implementation

- One issue that is yet to be addressed is the selection of the component responsible for event handling
  - The Source Monitor can handle the event after it is placed in the queue
  - The Processor can process the event as the first stage of its processing pipeline
- The nature of the Proactor pattern provides automatic event handling at the Source Monitor stage
  - To provide consistency between Proactor and Reactor implementations, the Reactor implementation should implement the Source Monitor as the event handler
    - The monitoring and event handling functions of the Source Monitor will be functionally decomposed such that each is implemented as a sub-component
    - The sub-components may be executed concurrently

Data Processing

- The Data Processor is composed of a set of one or more pipelines
  - Pipelines consist of a series of filters, transforms, and converters
    - The filter receives a const reference to a message and returns a Boolean value indicating the validity of the message; processing will abort if the return value is ‘false’
    - The transform receives a reference to a message and operates directly on the message, returning a Boolean value indicating success or failure of the operation; processing will abort if the return value is ‘false’
    - The converter receives a const reference to a message of type A and a reference to an empty message of type B, transforming the data from message A to the format expected for message B, returning a Boolean value indicating success or failure of the operation; processing will abort if the return value is ‘false’
  - The pipeline components are implemented as a hierarchy with the transform as an extension of the filter and the converter as an extension of the transform
  - Each pipeline is terminated by a consumer who defines how the data is used

Reform Data Processing

- The current Reformer syntax specification creates an artificial limit of one converter per pipeline
  - This limitation will be addressed with future modifications to the language syntax
- Two types of pipeline consumers are currently defined for the Reform language
  - A consumer to forward data from an input data source to an output data source
  - A consumer to accept a client connection from a server source, to handle inter-Reformer communication through data channels

Pipeline Construction

- Pipelines are used to connect a single input source with one or more output sources
  - A single input source is linked to *n* output sources with *n* pipelines
    - Allows modifications to be made independently to each connection
    - Each message received must be duplicated *n* – 1 times, providing a copy of the message to each of the *n* pipelines
      - Employ linear typing
    - Possibility that if each link is identical, they can be reduced to a single pipeline with *n* output sources
      - If one source requires a pipeline modification it will be separated to its own pipeline
      - Increase to overhead and complexity; not a likely choice for implementation

Pipeline Data Types

- The data types processed by the pipeline can classified with two categories
  - Message data received from a data source/stream
    - Message data is transformed and converted to conform to formats required by the destination
  - Client connections received from a server source
    - Client is a source encapsulated by a pipeline message
    - A series of filters and transforms may be used to perform any necessary handshake/connection negotiation

Data Type Restrictions

- Each pipeline sequence must operate on the same data type
  - Arguments for filters are specified as an abstract type, subject to immediate type checking by the filter
    - The Reform language will handle a type mismatch by throwing an exception and printing a message to the console
      - Remote management and notification may be complicated
  - The converter will produce a new data type
    - All filters and transform following a converter must process the new data type

Reform Data Types and Functions

- Formats, filters, transforms, and converters created through the Reformer exist as a collection of expressions
  - Expressions are submitted to the interpreter
    - The interpreter is accessed as a Singleton
  - Expressions are stored within a container class
    - It may be necessary to insert special references to function local variables which are provided to the container class by the interpreter
    - Prefix variables with a $

Pipeline Design

- The pipeline is designed to receive data from *n* input sources and provide data to *n* output sources
  - The Reform language is responsible for placing restrictions on the number of sources which may be simultaneously connected to a pipeline
  - Must evaluate the current pipeline concept defined for Reform
    - Examine the 1 input, *n* output language restrictions
    - Reform grammar does not have syntax for interacting with pipelines
      - Pipeline is implicitly defined when linking sources
      - Add pipeline construct to syntax
      - Add syntax for configuring pipeline
  - Explicit syntax for pipeline manipulation simplifies pipeline management
    - Possible to create a pipeline with *n* input and output sources
    - Easier for programmer to add and remove filters

Pipeline Management

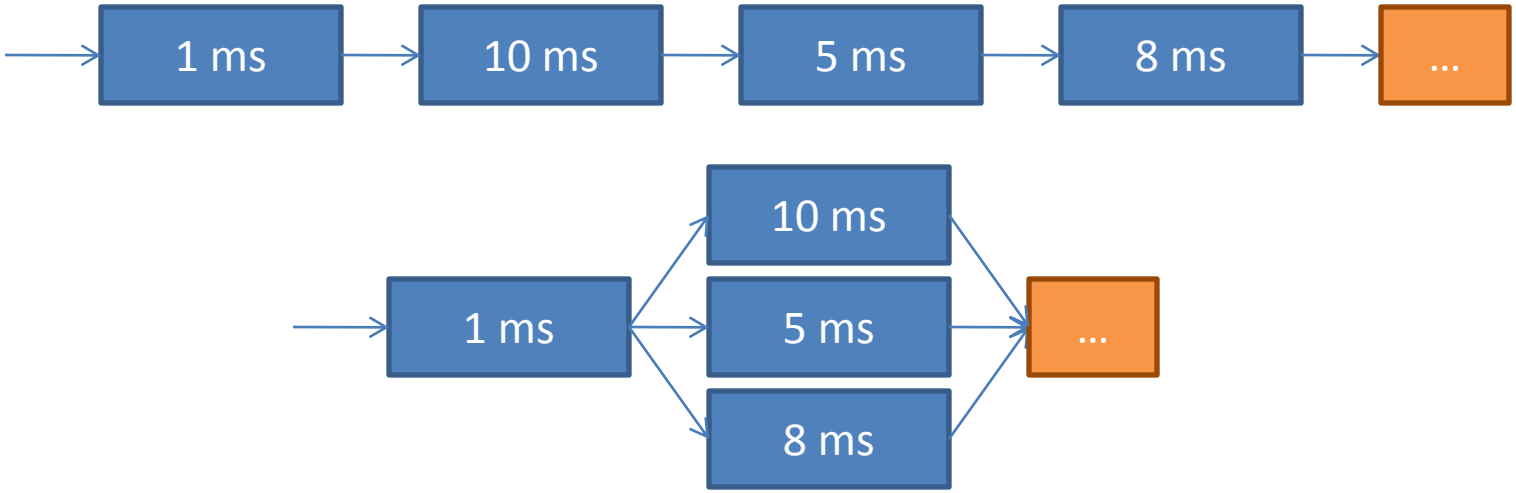
- Each pipeline has a queue for serialize data received from an input source
  - This is necessary for a system allowing *n* inputs to a pipeline
  - The Monitor adds data to the queue as it arrives and the pipeline Processor removes data for processing
    - The queue must be 'protected' for the Proactor case
      - OS is asynchronously dispatching data received from each source; each source is monitored independently
- Processed data is queued and a write request is registered with the Monitor
  - Queue must be single writer/reader protected
  - Data is sent when output source is ready to send
    - A single data item is transmitted for each write request
      - Requires *n* requests for *n* items

Pipeline Management

- Pipelines are grouped by input source through a dictionary keyed by source address
  - Provides Monitor with direct access to pipelines associated with source
  - Monitor retrieves pipelines for source and adds to data to the pipeline queues
- Processed data is queued and a write request is registered with the Monitor
  - Queue must be single writer/reader protected
  - Data is sent when output source is ready to send
    - A single data item is transmitted for each write request
      - Requires *n* requests for *n* items

Pipeline Optimization

- Filters are read only and may be applied in parallel
  - Applied in groups so that single short filter may precede a group of larger filters, etc
  - The following sequence with a minimum execution time of 1 ms and maximum execution time of 24 ms which may be reduced to a maximum execution time of 11 ms



Questions

- Does allowing multiple sources to share a single pipeline improve performance?
  - Possible impact on parallelism due to synchronization requirements
  - Reduction of memory footprint through shared resources
- Does allowing multiple sources to share a single pipeline increase overhead?
  - More effort for pipeline selection
  - Requires a pipeline queue
    - Also required if Monitor reads data from source, instead of pipeline reading data from source
- Is it necessary to differentiate between sources receiving data for a single object and sources receiving data for multiple objects?

Monitor Design Summary

- Monitor consists of two tasks
  - Task one waits for ready state from source(s); ready sources are added to a ready queue; monitoring for source is disabled
  - Task two processes source and submits result to pipeline; pipeline is accessed through a dictionary keyed by source address; enables monitoring of source
    - Dictionary may need to key by source address and format type to support the one format per pipeline/*n* pipelines per source paradigm
    - Evaluate complexity of multiple formats with a single pipeline
      - Probably one type per pipeline with support for polymorphic subtypes
  - Both tasks could be grouped together and executed by a single thread
    - Removes need for queue
    - Fits single processor design
    - Potential delay to monitoring of multiple high rate data sources
    - Something to evaluate

Pipeline Processing Design Summary

- Each Pipeline operates as an independent task
  - The Pipeline contains a data queue to be filled by the monitor for processing
  - Pipelines consist of one or more filter groups, transforms, and converters to process data
    - Subtasks process filter groups in parallel
  - Input sources provide received data to pipelines for processing
  - Output sources process data with pipelines before transmission
  - A Termination function determines what to do with data processed by the pipeline
    - Data received from input sources is setup for transmission to destination or submitted for internal processing
    - Data for output sources is transmitted
  - Pipelines are associated with sources through a dictionary structure for easy access
    - Must also associate with input type for sources receiving multiple types

Source Design Summary

- The Source encapsulates an OS specific data source such as a file or socket
  - The Source contains a HANDLE to the data source for monitoring
  - The Source contains a Classification object which maps the raw stream of bytes received from the source to a Format
    - Classifier looks for specific characteristics of and values contained within the message which indicate format type
  - A Source may receive data for a single object or for multiple objects
    - Possibly two source types for differentiating between single object and multi-object streams
  - The Source contains an optional Cache object for maintaining a finite data history
    - Separate cache objects for single object and multi-object streams