Pelagic Real-Time Platform Capabilities

Understand Real-Time Systems. In Real-Time.

July 27, 2017 [i]

www.FishEyeSoftware.com/Pelagic

Contents

Overview

Modern real-time systems have evolved to a complex cacophony of widely distributed, largely isolated systems. While these modern systems expand capabilities, they add a burden to interoperate, maintain, and analyze as a whole, counter to the need to reduce costs and personnel. Two aspects that are required to monitor and control such systems are: (1) access to data from disparate applications within these systems and (2) performance monitoring of the system resources themselves, including both operating system resource usages and collective software application performances. The availability of this data in real-time enables system data consumers to receive information relevant to their Community of Interest (COI) and system managers to understand the overall system state, manage various resources, and dynamically adapt applications and resources to changing loads and resource availabilities. The Pelagic Real-Time Platform™[ii] enables system developers, integrators, testers, operators, and maintainers to understand real-time systems throughout their life cycle. It allows these system stakeholders to understand what is happening as systems operate and in real-time.

FishEye Software's unique technology provides tools for monitoring and controlling systems that are widely distributed, complex, and operate in real-time. These tools offer the following features, functions, framework products, and services:

- Data collection schema definition and data distribution services enable users to identify which data in applications is to be captured, compose the data using domain-specific languages (DSLs), and then distribute the data in a self-describing form through an open standard publish-subscribe infrastructure.
- Open, standard data archive file formats: log the collected data in user-selectable formats, with built-in support for Hierarchical Data Format, version 5 (HDF5), an industry standard for data exchange in high-performance, real-time analysis applications.
- Automated generation of data collection schema: automatically search within application code products to catalog application data of interest.
- Data replay services: playback any selected archives for post-run analysis or for simulation purposes.

- Pre-publication data processing: enable end-users to specify logic to distill raw data collected by applications into reduced information prior to publication, using any of the following:
  - Data Filters: enable end-users to define selection criteria that reduce the volume of data that is published.
  - Complex Event Processing (CEP) enables end-users to evaluate and take action on real-time events.
  - Downloaded client algorithms: enable end-users to define algorithms in standard analysis tools (e.g., MATLAB, Python, Excel), specify bindings of data collection schema to their arguments and output values, download those specifications to a sourcing application, and have the associated processing be performed by the publisher prior to publishing the output values.

The benefits obtained by using these capabilities are:

- A Simple Way to Understand Complex Systems
- Manage and Control Large Real-Time Data Volume and Velocity
- Expose Internal Data, making it Open and Easily Accessible
- Reduce System Life-cycle Costs
- Move Post-Processing Analysis to Real-Time Analysis
- Real-time data distribution for remote system monitoring and support
- Highly scalable architectures through an open standard publish-subscribe infrastructure
- Run-time discovery of new data types (vice compile-time data type inclusion)
- Lower data throughput demand via higher-level events and server-side algorithms
- Improved understanding of system operation via user-defined domain-specific languages
- Reduced the cost of operation and maintenance personnel through automated decisions

Pelagic Real-Time Platform Capabilities

The following sections outline the capabilities of Pelagic products. Pelagic is designed as a flexible real-time microservice architecture consisting of directly deployable architectural mechanisms, a framework for constructing application-specific solutions, and a set of services for making those solutions available in a distributed computing environment.

1.1. Motivation

FishEye has long recognized the need for a set of tools for handling large datasets, specifically in the area of fusing sensor data from various sensors. This recognition stemmed from our contributions to several complex radar sensor development programs. While the tasks performed by these systems varied widely, they shared common aspects, including being hard real-time and requiring the ability to extract internal data for subsequent analysis.

Motivated by this need and the strong conviction that a flexible, standards-based, high-performance solution would make an enormous impact on the performance of complex real-time systems and the full life cycle productivity of the ecosystem, FishEye performed market research, developed a business plan, and launched demonstration development of the Pelagic[iii] in early 2008. This technical design and development work continues in parallel with the business development effort.

1.2. Background

Modern distributed systems comprise many integrated and discrete complex software systems that run mission-critical, real-time processes. Some legacy systems are being upgraded with modern technology in order to extend their life (and investment payback) for several decades. Other legacy systems are considered "untouchable"; yet, it is desirable to somehow make them interoperable with the newer, fully integrated systems. New systems have the opportunity to embrace the newer technologies, yet should be architected to be open, evolvable, and scalable.

While these integrated distributed systems typically perform widely disparate tasks, they all share a common underlying need: namely, the ability to monitor and distribute information about the system's internal operation in real or near real-time without perturbing the performance of the system's primary task. This need to monitor and distribute information is motivated by the following:

- **System development** – During system development, engineers require the ability to observe the internal operation of the various modules that comprise the system, ensuring they are performing according to system requirements. Designers and developers also need to understand and account for system usage and overhead constraints.
- **System test** – During Formal Qualification Testing (FQT), it is difficult to ensure correct operation of the entire system strictly through the observance of the external behavior of the system. Typically, FQT test plans include the requirement to observe the correct *internal* operation.
- **System optimization** – Deployed systems may require automated, semi-automated, or manual optimization of various operating parameters based on environmental conditions or mission tasking. The ability to record internal data for subsequent offline analysis is often required to perform effective system optimization.
- **System operation** – With the trend toward developing modern, scalable, and real-time systems using standard off-the-shelf hardware and operating systems, the ability to move processes among platforms to account for load variation and possible subsystem failures necessitates the real-time collection and provisioning of a variety of system metrics and events.
- **Post-mission analysis** – It is crucial to accurately record and analyze both simulated and real mission data to provide feedback to both the engineering and operational teams.

Historically, the need for non-intrusive data capture and distribution was addressed on a program-by-program basis. Program budget and resources are dedicated to developing a custom solution for the individual program. As a result, the technology for accomplishing real-time data capture and distribution is reinvented for each program, creating a legacy of systems, each with its own data capture approach and generating its own proprietary, non-transferable data products. Clearly, having data capture technology tied to a specific program results in solutions that run the risk of becoming brittle, non-scalable, non-maintainable, or obsolete. Proprietary, non-standard data formats also increase the cost of data analysis tools and hinder the ability to add future capabilities that utilize that data.

It is estimated that upwards of $10B is spent annually on real-time system development. Presently, there is a lack of commercial products to address the data capture and distribution problem that each of these systems faces. The product set described herein intends to fill these gaps. The development effort started in the fall of 2008. Some of these capabilities have been implemented as part of internal development efforts, while others have been developed as part of a Small Business Innovation Research (SBIR) project. Additionally, some capabilities have been proposed as part of two separate SBIR efforts, and others are included in FishEye's product roadmap[1].

Concept of Operation

1.3. Actor Diagram

Figure 1 shows a UML actor diagram that establishes the system boundaries for the Pelagic products and the roles played by the various external entities that interact with the Pelagic products. The primary points of interest in this actor diagram are that the Instrumented Resources are only minimally impacted by the Pelagic instrumentation and that Pelagic serves as the bridging technology to extract data from those resources and deliver it in forms suitable for a range of processing targets, both in real-time and as logged data.

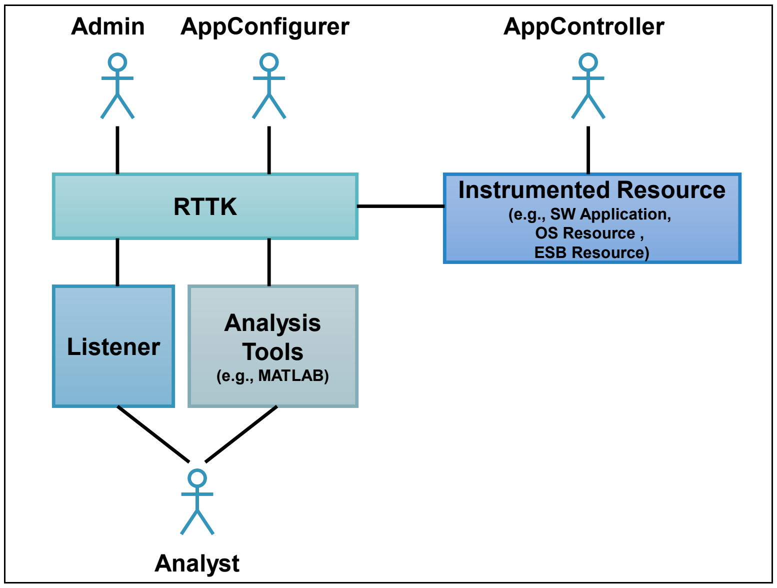


Figure 1. Actor Diagram for Pelagic

The responsibilities of the actors are summarized in Table 1. The primary point of this table is that the sole responsibility of the "Instrumented Resources" is to log their data, and that Pelagic handles how the data is to be pre-publication-processed and distributed to subscribers.

Table 1 – Actors and Responsibilities

Actors	*Responsibilities*
Administrator	Identifies authorized users to Pelagic. Configures Pelagic for usage by Application Configurers and Application Controllers.
Application Configurer	Configures Pelagic for one or more monitored resources.
Application Controller	Configures the Applications for usage with Pelagic and controls the running of those applications.
Instrumented Resource	Logs Data to Pelagic.
Listener Application	Subscribes to and receives real-time application data.
Analysis Tools	Receives Pelagic data artifacts.
Analyst	Configure and observe near real-time and post run data outputs.

1.4. Operational View

While not in scope for this white paper, the use cases that were established for this set of actors are suppressed. Instead the following representative operational steps for the “happy path” usage pattern should suffice for conveying a fundamental concept of operation:

1. An Administrator sets up accounts for qualified users of Pelagic products and services.
2. An ApplicationConfigurer (in a system engineering role) uses Pelagic to analyze an application to identify the application class definitions from which he intends to collect data and defines (automatically or manually) a catalog of data schema definitions (a MetaLog) into which the data values will be copied. These data collection schemas will be referred to as “Fundamental Data Artifacts” (FDAs) and will be used as the basic building blocks for all the functionality of Pelagic. Because these FDAs constitute meta-

information about the application classes, the catalog in which they are maintained will be referred to as a “MetaLog”.

3. An ApplicationConfigurer (in a developer role) instruments the application with data collection calls for FDAs that have been defined in the MetaLog. For legacy applications that already perform data collection, these Pelagic-directed calls can be included either as additional substeps within the existing data collection function or as replacements to the existing data collection functions.
4. An ApplicationConfigurer (in an integrator role) defines a “project”, wherein a “project” identifies which FDAs will be collected from which application runs and which pre-publication data processing is to be applied.
5. Prior to starting a run or during a run, one or more Analysts start their listeners and configure them for receiving the data streams of interest by subscribing to the associated FDAs and submitting-or-enabling any pre-publication client-supplied data processing algorithms.
6. An ApplicatonController sets up the project in preparation for making a run, by staging the FDA configuration settings and starting the execution of the participating application programs.
7. During the run, the runtime Analysts view the application and/or OS-level resource monitoring agents.
8. At any time after the run, an Analyst can request a playback of a previously-logged session, reconfigure the listeners with differing FDAs, filters, or pre-publication data processors and observe the new set of outputs. For example, an Analyst may choose to revise a MATLAB script to see if it produces more useful analysis. Or, an Analyst may choose to define different Complex Events to try to improve the situational awareness of the observation perspectives.

Notional Architecture

Based upon the use cases noted in the previous section, an object-oriented analysis was performed, the design forces were introduced, and an object-oriented design was developed to carry those forces for the given requirements. That design has been maintained as the basis of the software implementation. A notional architecture of the resulting current design is shown in Figure 2.

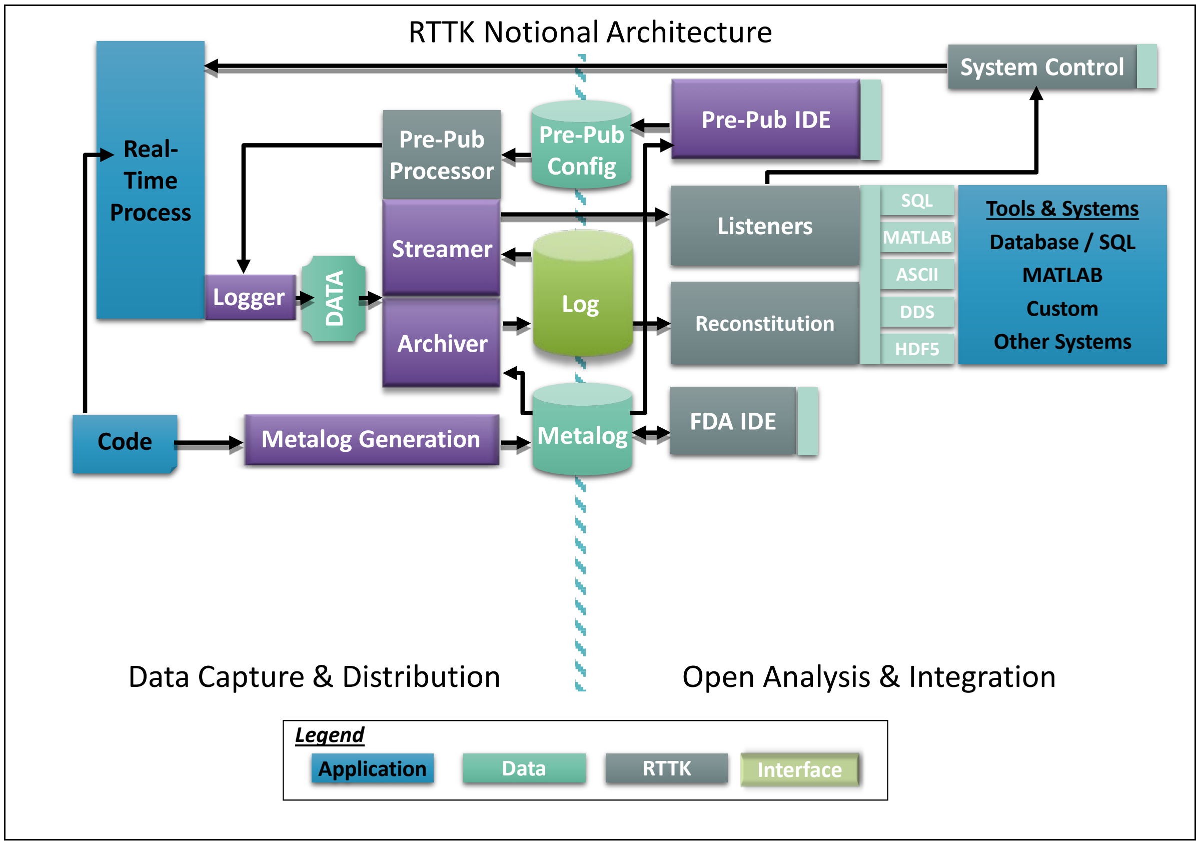


Figure 2. Pelagic Notional Architecture

Pelagic manages FDA data representations in a MetaLog, which serves as a common repository for data schemas, facilitating archiving and streaming of datasets. The MetaLog can be populated manually or by a MetaLog generation tool. The MetaLog generation tool analyzes application source code products for candidate data collection types and then automatically populates the MetaLog. Instrumentation of application processes is done by adding simple log function calls with arguments that reference data-bearing objects in the running application program. As data is collected, it can be both logged to disk by the Archiver and streamed to Listeners. Pelagic utilizes Hierarchical Data Format, version 5 (HDF5), as its logging format due to its wide industry acceptance, hierarchical characterization capability, low storage footprint (in binary), high read/write speeds, and platform independence.

It is recognized that some clients may not find an existing set of FDAs in the MetaLog that meets their needs. For example, warfighters at the “tactical edge” typically have very small bandwidth communications devices that cannot handle large data throughputs. Even with adequate communication devices, subscribers may want to perform some automated data processing on available data on the server side before publishing information derived from collected datasets. A solution for both situations is to provide a capability to pre-process the datasets with logic that distills the data into a smaller set of more directed information. In some cases, the nature of this data pre-processing may align with the CEP transformation capability. To support these cases, Pelagic will offer a server-side CEP engine as a service. In cases where the CEP treatment is not well-suited, Pelagic is planning to provide a service for client-defined algorithm support, where clients define their algorithms in industry-standard analysis tools (such as MATLAB), use an “Algorithm IDE” to browse the MetaLog for the FDAs that will serve as the arguments to the scripted program, find or define FDAs that will carry the outputs of the scripted program back as published FDAs, and download the scripted algorithm with FDA bindings to the publishing server. As data is streamed from applications, the selected FDAs will be passed through the algorithm, and the generated FDAs will be published as well.

Further details of the functional capabilities of the Pelagic tools, frameworks, and services are provided below.

Functional Capabilities

This section provides brief descriptions of Pelagic's functional capabilities. The sections are ordered from basic definitions of data types to increasingly more sophisticated tools, frameworks, and services.

1.5. Data Schema Definition

The basic unit of manipulation in Pelagic is called a Fundamental Data Artifact (FDA)[2]. An FDA is a data record consisting of a unique identifier[3], execution context data, and a data payload. The execution context is information about the process from which the data has been collected, for example, time, process id, and thread id. Pelagic provides a default definition for the execution context data, but users can choose to replace the default execution context data with their own definitions including the ability to have the Pelagic logger call back to a user-defined function to populate the execution context portion of the header. The data payload portion of the FDA is defined by a Data Schema, which is a sequence of Data Members of user-specified types, including any combination of primitives (for example, int, float, char) and aggregates (for example, array of primitives, sequence of data members, aggregates of aggregates). The logical specification of what constitutes an FDA definition is considered “data about the data”, often coined by the word “metadata”. This metadata is maintained in a version-controlled database, called a MetaLog. The left of Figure 3 shows a conceptual illustration of what an FDA definition looks like in a MetaLog. During application execution, log calls in the program extract data

[2] An FDA corresponds with entities in other programs such as Data Collection Records (DCRs), Logical Record Ids (LRIDs), Diagnostic Trouble Codes (DTCs), etc.

[3] The unique identifier of an FDA contains (at a minimum) a unique integer and a version identifier.

values from the application and send them to a Logger. The Logger combines the FDA identification data, execution context data, and payload data to form a data collection record, as illustrated in the right part of Figure 3. This figure illustrates only the case where Data Members are of fixed size, however, the actual Pelagic toolset can handle variant records (as in Ada).

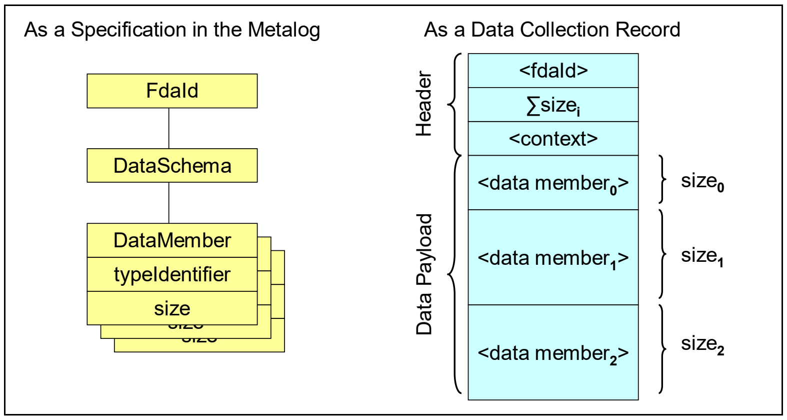


Figure 3. Conceptual Model of a Fundamental Data Artifact (FDA)

1.6. MetaLog Creation

A MetaLog can be populated programmatically through an Application Programmer Interface (FDA API), interactively through an FDA Interactive Development Environment (FDA IDE), or automatically though the MetaLog Generation Tool (MetGen). In all cases, the definitions of the application classes from which data are to be collected must be available for setting up the Data Member definitions of the Fda Data Schema. For the FDA IDE and the FDA API, it is sufficient to have just the logical specifications of the application data types; however, actual source code facilitates the creation of the Data Schema. For the MetGen tool, access to the source code is required. The initial version of MetGen can analyze Ada code. MetGen for C++ code is in progress.

The logical process for defining an FDA is the same for all three techniques and is illustrated in Figure 4. First, the information needed for system monitoring and control is congealed into a set of logical data collection record definitions. Each logical data collection record definition is expressed as a DataSchema in Pelagic. A type, “T_i”, is defined for the DataSchema and is registered in the MetaLog registry. Details of this process are suppressed from this white paper: suffice it to say that the information in “T_i” is used to move dataFields between storage spaces, for example, from an octet sequence into HDF5 files. Each DataSchema is assigned to a unique FDA in the MetaLog. For each FDA, the definitions of the application data types (e.g., classes and structs), the data is extracted from their code-based products. Next, for each data member in the Application Data Type that is to be collected, a semantically equivalent Data Member is added to the Data Schema of the FDA. In the simplest case, an entire instance of the application data type is mapped into a Data Member of equivalent type in the FDA. In the general case, a sequence of selected sub-fields of the application data type is mapped to a sequence of equivalent type fields in the FDA[4].

[4] This case has been coined “cherry picking” and is not yet implemented in Pelagic.

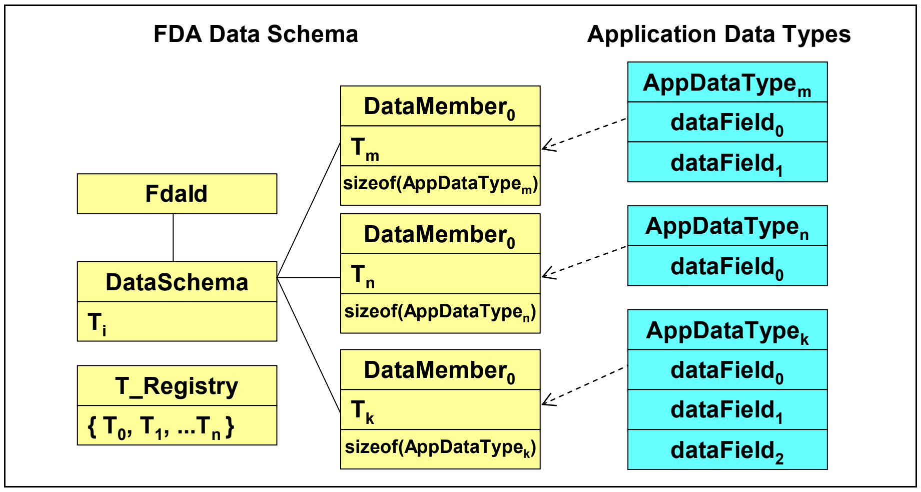


Figure 4. Mapping Application Data Types to FDA DataSchema

1.7. Data Collection

Data collection with Pelagic can be done from applications as well as from the OS-level resources in which the applications are executing.

1.7.1. Application Data Collection

In order to collect data from an application, its source code must be instrumented with logging calls, referred to herein as “dataRecord” calls. These calls can be placed anywhere in the code where the values from application objects are available. However, if one is instrumenting an application that already has logging calls of its own, the Pelagic “dataRecord” call can be inserted within the existing logging method definition *alongside* the existing data extraction code or *instead of* it. This allows application developers to use Pelagic “dataRecord” to “shadow” an existing data collection mechanism either to verify its performance until a “cutover” decision is made to switch to using Pelagic or to “tee” the data to BOTH record to the existing mechanism AND to augment the system with additional features available in Pelagic that do not exist in the legacy system.

The interface to "dataRecord" is generic and introduces no compilation or link dependencies on the application code base[5]: "dataRecord(int fdaId, void * src0, int size0, …, void * srcN, int sizeN)". Figure 5 illustrates the process of data collection. When the "dataRecord" function is called, the memory addresses of the values of the application data types are passed in as arguments, "srcX". For each application value, the memory size of the value is also passed in, "sizeX". The "dataRecord" function calculates the amount of memory required to record the Header-plus-Payload, based on the data collection configuration settings and the sum of the sizes passed as arguments. Then it obtains the required storage area as a "chunk" from the collection buffer memory, fills in the header part with the FDA value, and the execution context information. Then "dataRecord" traverses the list of "srcX/sizeX" pairs, copying bytes from application memory to the collection buffer. This "chunk" is now available for further processing, such as pre-publication processing, streaming and archiving.

[5] Some data recording systems use a strongly-typed collection function that exposes application data types to the "dataRecord" call. In such systems, when a new application data type is added to the set of data collection records, the application must be rebuilt, in some cases causing extensive recompilation.

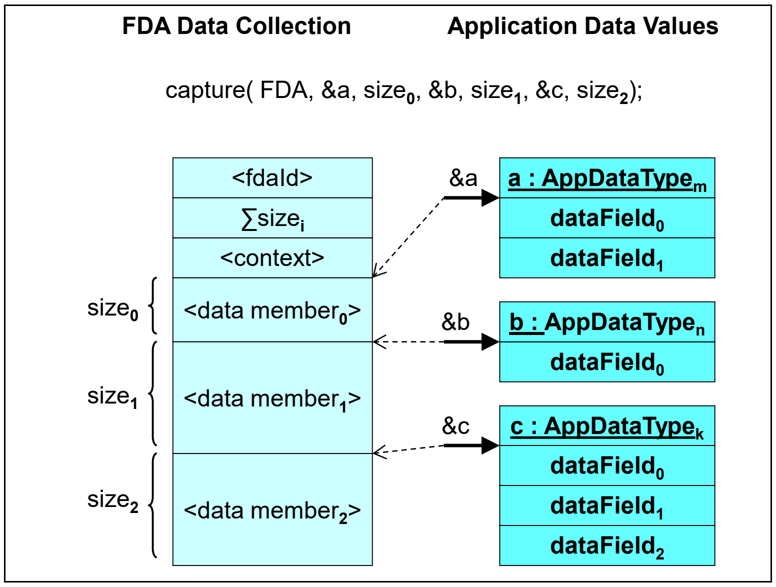


Figure 5. Collecting Values from Application Data Types

Prior to running an instrumented application, the collection strategy for that run must be established. This can be done using a project run configuration tool to create a configuration file (XML) or directly creating/modifying the configuration in a standard editor or using a combination of the two approaches. This file can contain a run identifier, the collection priorities of FDAs that are to be collected for that run, any filters to be applied prior to collection, and the dispensation choices for collected data. The run identifier can be used to coordinate the FDAs in this program execution with other processes, for example, remote subscribers. The priorities range from OFF to ESSENTIAL and are used during the run for shedding the collection load when the collection buffer resources become successively degraded. The filters cause the collection of an FDA to depend upon contextual information, such as value thresholds and time windows. The dispensation choices indicate what to do once an FDA has been collected, causing the FDA to do any of the following: buffer it in the application, publish it as a stream. When the dispensation choice is to buffer it in the application, FDAs are collected in application memory until the volume of collected data reaches a flushing threshold, at which time a local Archiver is signaled of the buffer’s availability and the Archiver writes the buffer contents into a HDF5 log file. When the dispensation choice is to publish the FDA as a stream, the Logger immediately publishes the FDA using DDS so that subscribers can receive FDA data as it is collected.

The final requirement to incorporate Pelagic data collection into an application is to link the Pelagic Logger library into the executable, as indicated by the "Logger" modules in the notional architecture in Figure 2. The Logger module, at program startup, will read the data collection configuration file. Based on the settings in the configuration file, the Logger initializes the FDA runtime settings. If there is a dispensation choice to publish FDAs, the Logger may set up the DDS publishing facility and publish the FDA/DataSchema as DDS Topics. If there is a dispensation choice to buffer FDAs, the Logger initializes the memory buffering mechanism and creates a local Archiver with its own threads to perform the writing of HDF5 log files.

It should be noted here, that a user can choose to use a remote Archiver instead of a local Archiver by setting the buffering dispensation choice off and the publishing dispensation choice on and then subscribe to the FDA as a Content-Filtered Topic, filtered by the run identifier that was used to configure the Logger. Pelagic provides such a standalone Archiver though its SOA to act as a standalone subscriber.

1.7.2. Data Streaming

When a Logger has been configured to perform streaming of its FDAs, at startup it initializes the entities in the DDS publish-subscribe network programming framework (domain participant, publisher, data writer, and topics) based on the configuration settings. One of the settings in that file identifies the DDS QoS file to use in the given run. Whenever a "dataRecord" request is made from the application (or RM Agent), if the given FDA is configured to be published, it is published through the DDS network fabric.

1.7.3. Data Archiving

Data archiving is defined herein to mean the writing of data to a log file. In Pelagic archiving is performed by an Archiver object. An Archiver object can write the data in either native binary format or in HDF5 format. HDF5 has been selected for Pelagic due to its wide industry acceptance, its hierarchical characterization capability, its low storage footprint (in binary), its high read/write speeds, and its platform independence. Some users may choose the binary format to be backward-compatible with other data collection schemes, however new programs will likely opt for writing the data in HDF5 format.

Archiver objects can be collocated with the Logger in the application program (local) or they can be created in a separate process or computer (remote).

When the Archiver is local, the Pelagic startup code creates an Archiver object, provides it with a specified number of threads, and connects it to the Logger. As the Logger fills its buffers with data collected from its application, it signals the Archiver that a data collection buffer is ready to be written to disk.

For remote Archiving, there are two options. One option is to use the ready-built standalone Pelagic Archiver product that extends the FDA Listener to automatically create the stream-in interface of the FDA Listener to get FDA data from the application. As data is streamed-in, it is written to the specified log file in binary or HDF5 format.

1.7.4. Playback Services

Once a system has run for a period of time and has logged its FDA data, Pelagic provides services for playing back any selected sets of FDAs from the log files for any selected duration of time. As illustrated in Figure 2, this data is formatted into its native streaming format and passed to a Streamer. Since the Streamer is agnostic to its buffering sources, it streams the data just as if it were originating in running applications, pushing it through its normal publishing channels.

1.7.5. Pre-Publication Data Processing

It is widely recognized that modern systems can emit orders of magnitude of raw data than is strictly needed by all subscribers. Not only does this place additional processing and throughput burdens on the system, but also does is result in information overload on its human consumers. A repeated request from users has been to be able to perform various sorts of preprocessing on data and to offer alternate, distilled data that is targeted at more specific views that indiscriminate subscriptions. To this end, several capabilities are available: data filtering, Complex Event Processing, and Client-Defined Algorithm Download.

1.7.6. Data Filtering

Users can specify a rich range of filtering criteria such as start time, stop time, threshold values, recording frequency, recording triggers, etc. to be applied in data collection during a project run. These settings are specified in the FDA configuration file (XML) that is read in by the Logger at the startup of each application. The settings can be globally applied to all FDA in that run as well as on a per-FDA basis. As data is considered to be collected from an application, these collection criteria are first checked before they are collected-and-published. For example, users can request that a particular FDA not be collected until it has reached a given threshold value or only when its value is within threshold limits. Or a FDA be collected no more frequently than a specified period. Or a FDA not be collected prior to a given start time nor after a given stop time.

1.7.7. Complex Event Processing (CEP)

An <u>event</u> is defined as something that happens at a point in time. In the context of Pelagic operation, FDA publication can be viewed as a simple event. Figure 6 illustrates an example where the onboard monitoring system of a car may subscribe to data from several different sensors: a sensor that indicates whether the driver is in the seat and a sensor that indicates the current speed. Each sensor publishes the states of these sensors periodically. An FDA Listener in the onboard computer might observe that the driver has left the seat and the speed is above 10 mph. In this example, the runaway car notification is a Complex Event, that is, a higher level event that is inferred from the occurrences of lower level events. Complex Event Processing performs operations on complex events, including reading, creating, transforming, and making inferences on them.

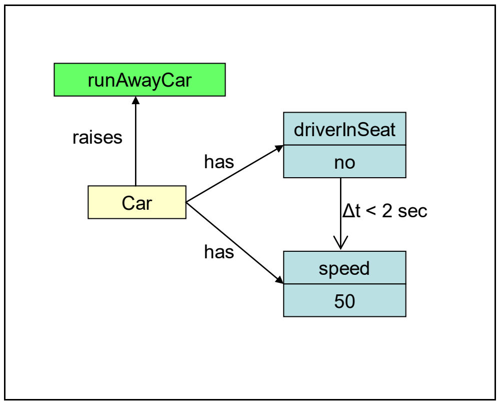


Figure 6. Illustration of Complex Event

Note that because the trigger rules are specified in the language of the application domain as opposed to being specified in a CEP-specific language, users and system engineers can readily understand the CEP trigger rules without needing to learn the language of a specific CEP tool.

CEP Inference Engine. The CEP Inference Engine subscribes to the FDAs that can serve as events. As FDA values are received, the CEP Engine determines what "IF" clauses are triggered, performs the execution of the inference rule to create the Complex Event, adds that Complex Event to its assertion set, and publishes the FDAs associated with the "THEN" part of the inference rule.

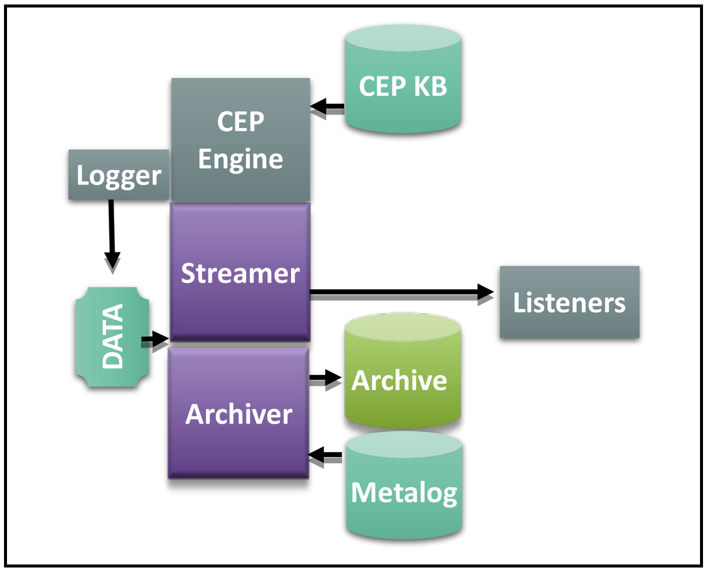


Figure 7. Complex Event Processing

Example Application

An example distributed system has been devised to illustrate the functionality that has been accomplished to date. **Figure 8** illustrates data interactions between a Radar application, a Command and Control application, and a Resource Management application. Data entities within the Radar application include target information and a track processing rate metric. Pelagic™ exposes these application data entities, storing information about them in a MetaLog. The Radar is able to publish either of these data entities by invoking a call to an Pelagic™ Logger library function, with parameters that provide an id (identifying the particular data entity), the address of the data entity, and the size of the data. The data, along with the id and a small amount of execution context information, is published on the distribution bus as a stream of octets. Pelagic™ uses the MetaLog information for the given id to provide the domain-specific semantics for this octet stream. In **Figure 8**, this is shown through having one type of data (target data) published for use by Command and Control while having another type of data (track processing rate) published for use by Resource Management. Because the data is published as an octet stream, any variety of data entities can be published without needing to create a separate interface specification, such as DDS IDL, for each specific data entity. The listener within Command and Control or Resource Management receives the id, execution context information, and application data, and applies the domain-specific context to the octet stream.

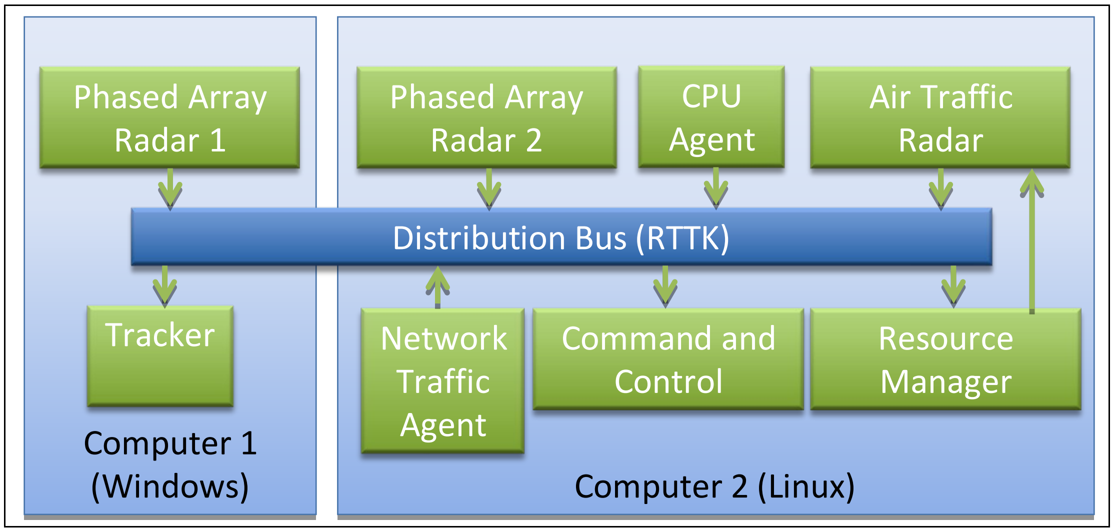


Figure 8 Enabling Dynamic Adaptive Resource Management

In this example, two Resource Monitoring (RM) agents (a CPU Agent and a Network Traffic Agent) were produced as well a (simulated) phased array Track Production Rate performance report FDA. The RM Agents demonstrated how strongly typed data on OS-level resources could be captured in non-intrusive, standalone daemons. The Track Production Rate FDA illustrated how the existing Pelagic^{TM} application data capture mechanisms are sufficient to extract meta-data from applications so that the applications themselves can be viewed as networked resources by Resource Monitors.

A variety of scenarios, including failure scenarios, were employed to demonstrate the ability to collect and publish system and application performance data. The “Memory Failure” scenario illustrated that the CPU Agent showed increasing values of stack space in use as the stack transitioned to an overflow state, and increasing values of process virtual memory size as the heap reached a state of exhaustion. The "Link Failure" scenario illustrated that the Traffic Agent showed a sudden stoppage of packets on the associated network interface when the Ethernet connection was removed, and additionally demonstrated the ability to instrument an application to publish performance data. These results validated the proposition that real-time data collected via Pelagic^{TM} provides a good basis for developing Event Models for use in CEP.

Example Code

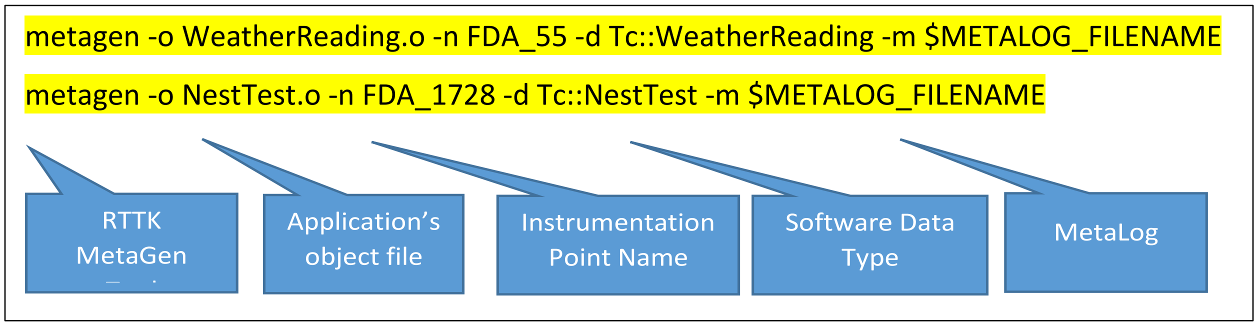
The incorporation of Pelagic into a system is through a two-step process.

1. Extract Metadata - Identify data types for runtime capture and extract schema with MetaGen.
2. Instrument the Application - Within your Application, Start Pelagic and insert Data Instrumentation Points

The following example shows the two-step process.

Step 1 – Extracting Meta Data with MetaGen

Figure 9 shows the process of using Pelagic's MetaGen to extract application MetaData.



```
metagen -o WeatherReading.o -n FDA_55 -d Tc::WeatherReading -m $METALOG_FILENAME
metagen -o NestTest.o -n FDA_1728 -d Tc::NestTest -m $METALOG_FILENAME
```

Figure 9 – Example Showing Pelagic MetaGen

Key : -o (object file defining the datatype structure);

-n (optional user number uniquely identifying the datatype to be captured);

-d (datatype identifier as specified in source code)

-m metalog filename

Step 2 – Instrumenting the Application

Within the application start Pelagic and insert Instrumentation by passing your Application to Pelagic “Capture” function. The Pelagic include file declaring the capture function in any translation unit (code file) capturing The Pelagic library statically linked into the application. The example currently shows the optional user defined identifier. The real identifier comes from code and is human readable, for example “namespace::classname”.

Figure 10 below shows an example program starting Pelagic and recording data. Figure 11 and Figure 12 show the code for some example data types. Figure 13 shows some sample output that was archived in an HDF5 log file.

```
#include "WeatherReading.hxx"

#include "NestTest.hxx"



#include "RttkStartup.hxx"

#include "RttkCapture.hxx"



int main(int, char**)

{

    int status = 0;



    // Use MetaGen here to extract and datatypes at run time.



    // Start the RTTK library, passing in configuration XML file which

    // associates the captured datatypes with publishers
```

Figure 10 – Example C++ Main Showing Calls to Pelagic

```
class WeatherReading

{

public:

    WeatherReading();

    WeatherReading(double time, int temp, int pressure);



    void print(std::ostream& os) const;
```

Figure 11 – Datatype WeatherReading

```
    class KeyLookup // nested within NestTest as a data member
    {
    public:
        KeyLookup();
        KeyLookup(int hash, int clash);
        ~KeyLookup();
        void hash(int value);
        void clash(int value);
        int hash() const;
        int clash() const;
    private:
        int hash_;
        int clash_;
    }; // class KeyLookup

    enum class NestTestArrayDims : int
    {
        MAX_INT_ARRAY = 4,
        MAX_DOUBLE_ARRAY_1D = 5,
        MAX_DOUBLE_ARRAY_2D = 6,
        MAX_FLOAT_ARRAY_1D = 7,
        MAX_FLOAT_ARRAY_2D = 8,
        MAX_FLOAT_ARRAY_3D = 9
    }; // enumerate demonstrated array data member lengths

    class NestTest
    {
    public:
        NestTest();
        ~NestTest();
        int get_found() const;
        void set_found(int value);
        KeyLookup get_keyLookup() const;
        void update(int updateKey);
    private:
        int found_;
        KeyLookup keyLookup_;
        int intArray1D_[(int) NestTestArrayDims::MAX_INT_ARRAY];
        double doubleArray2D_[(int) NestTestArrayDims::MAX_DOUBLE_ARRAY_1D]
                             [(int) NestTestArrayDims::MAX_DOUBLE_ARRAY_2D];
        float floatArray3D_[(int) NestTestArrayDims::MAX_FLOAT_ARRAY_1D]
                           [(int) NestTestArrayDims::MAX_FLOAT_ARRAY_2D]
                           [(int) NestTestArrayDims::MAX_FLOAT_ARRAY_3D];
   }; // class NestTest
```

Figure 12 – Datatype NestTest

```
HDF5 "datastore.h5" {
GROUP "/" {

   DATASET "Tc::WeatherReading" {
      DATATYPE  H5T_COMPOUND {
         H5T_IEEE_F64LE "time_";
         H5T_STD_I32LE "temperature_";
         H5T_STD_I32LE "airPressure_";
      }
      DATASPACE  SIMPLE { ( 2 ) / ( H5S_UNLIMITED ) }
      DATA {
      (0): {
            0,
            -1000,
            2000
         },
      (1): {
            100.001,
            -999,
            2001
         }
      }
   }

   DATASET "Tc::NestTest" {
      DATATYPE  H5T_COMPOUND {
         H5T_STD_I32LE "found_";
         H5T_COMPOUND {
            H5T_STD_I32LE "hash_";
            H5T_STD_I32LE "clash_";
         } "keyLookup_";
         H5T_ARRAY { [4] H5T_STD_I32LE } "intArray1D_";
         H5T_ARRAY { [5][6] H5T_IEEE_F64LE } "doubleArray2D_";
         H5T_ARRAY { [7][8][9] H5T_IEEE_F32LE } "floatArray3D_";
      }
      DATASPACE  SIMPLE { ( 2 ) / ( H5S_UNLIMITED ) }
      DATA {
      (0): {
            1001,
            {
               1,
               10
            },
            [ 0, 1000, 2000, 3000 ],
            [ 0, 100, 200, 300, 400, 500,
               10000, 10100, 10200, 10300, 10400, 10500,
               20000, 20100, 20200, 20300, 20400, 20500,
               30000, 30100, 30200, 30300, 30400, 30500,
               40000, 40100, 40200, 40300, 40400, 40500 ],
            [ 0, 10, 20, 30, 40, 50, 60, 70, 80,
               100, 110, 120, 130, 140, 150, 160, 170, 180,
               200, 210, 220, 230, 240, 250, 260, 270, 280,
               300, 310, 320, 330, 340, 350, 360, 370, 380,
…
               6601, 6611, 6621, 6631, 6641, 6651, 6661, 6671, 6681,
               6701, 6711, 6721, 6731, 6741, 6751, 6761, 6771, 6781 ]
         }
      }
   }
```

Figure 13 – Hdf5 Output from h5dump

Questions and Answers

What does MetaGen need to extract metadata?

Q: Does MetaGen depend on using extra information in the binary designed for use by a debugger? Does the code have to be compiled with a special compiler?

A: Yes, the technology uses compiler artifacts (symbols and structures some of which a debugger might use). It also extracts some info about the hardware platform (e.g., endianness).

The Pelagic Metagen does require a "-g" flag to access some of the info it needs (like use define names) from the object files (.o). If you don't want your executable (.exe) to be compiled with "-g" you can compile a ".o" file with and without the "-g" and then link your executable without the "-g" and let MetaGen extract info from the other ".o". Regardless the metadata extraction is automated and the real-time archive of data + schema is available.

Summary

Pelagic is a platform that provides a set of tools, services, and frameworks for monitoring and controlling real-time distributed systems. It enables system developers and integrators to gain a white-box view into the states of applications and operating system resources in real-time. It provides system maintainers with powerful and extensible introspection tools for troubleshooting problems. It provides system managers with tools for dynamic, adaptive resource management, enabling system optimization and enhanced survivability. It provides end-users with an open, scalable, and distributed architecture that is observable in the domain language of the end-user and allows user-defined extensions, such as complex events and user-specified algorithms. Pelagic transforms a cacophony of complex, isolated systems into an orchestrated ensemble of interoperating information sources tuned to the domain languages of its users.

SBIR Data Rights

Pelagic Real-Time Platform Road Map

Planned Capabilities

1.8. General Capabilities

- System monitoring and control: enable domain-experts and end-users to analyze and modify application monitoring and control strategies in their domain-specific language (DSL), without requiring software (re-)programming.
- Ontology-based information exchange services: enable a Community of Interest (COI) to compose higher-level information from basic application data using the semantics of the COI domain, specify information gathering rules based upon their ontology, and have the information published as standard Web Services data.

The benefits obtained by using these capabilities are:

- Improved situational awareness in the languages of the various COIs via standardized ontologies
- improved system survivability through dynamic, adaptive resource management

1.8.1. Notional Architecture

Based upon the use cases noted in the previous section, an object-oriented analysis was performed, the design forces were introduced, and an object-oriented design was developed to carry those forces for the given requirements. That design has been maintained as the basis of the software implementation. A notional architecture of the resulting current design is shown in Figure 2.

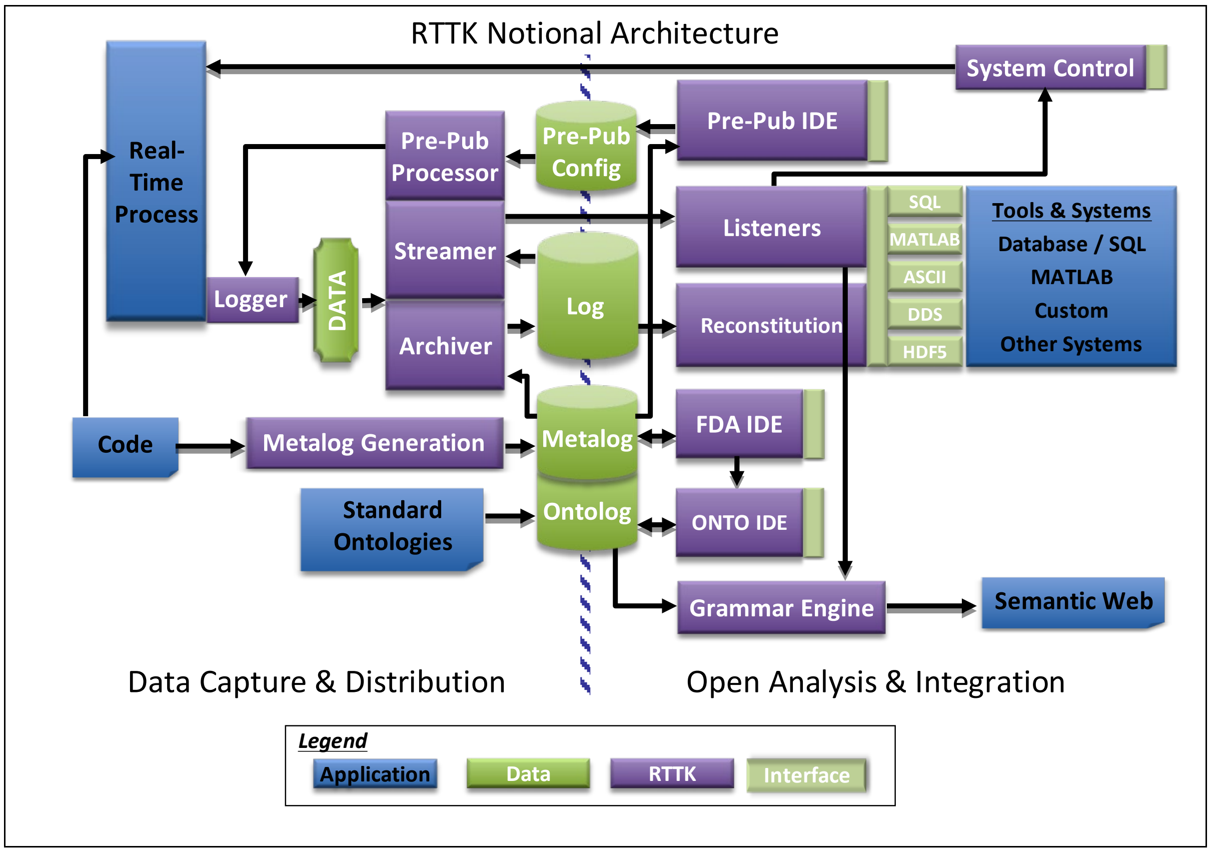


Figure 14. Pelagic Notional Architecture

Data Distribution Services (DDS) has been incorporated into Pelagic as the streaming technology. The collection of OS-level data can be done using (non-application) Resource Monitoring agents (RM Agents).

1.8.1.1. OS-Level Resource Monitoring Agents (RM Agents)

In addition to collecting data from applications, integrators and system maintainers may also need to monitor the underlying operating system resources for their performance (e.g., memory usage, paging, network traffic, etc.) in order to detect resource degradations and failures or to understand resource loading patterns imposed by the running applications. Having access to such data in real-time allows system maintainers to anticipate resource degradation and allows integrators to configure their applications to more optimally use the operating system.

The process for developing RM Agents is essentially the same as the process for instrumenting an application with data collection, except there is no pre-existing application providing the data sources. Rather a standalone “application”, referred to as an RM Agent, is devised that packages the code that queries the OS-level resources for metrics of interest (periodically, say), and records this data into an FDA that has been defined specifically for the appropriate data types. Similar to applications, the RM Agent contains a Logger that is configured using a configuration file. Pelagic provides such standalone RM Agents through its SOA to act as standalone publishers. At this point, Pelagic offers two RM Agents, a CPU Agent and a Network Traffic Agent. RM Agents can have additional configuration file entries beyond the FDA collection controls. For example, the CPU Agent can read the run identifiers to indicated which processes from which to collect metrics and the Network Traffic Agent can read the IP sockets that it is to monitor. There are plans to add more RM Agents, such as Thread Monitoring Agent, Disk Usage Agent, etc.

Since RM Agents are dedicated to a single application data type and do not need to be open to extension as do application data types, Pelagic has proves a strongly-typed interface in addition to the generic FDA interface for non-FDA subscribers (i.e., clients who are interested strictly in system monitoring, not application monitoring). These interfaces are expressed in IDL for clients to compile into their applications using the classic DDS publish-subscribe strategy.

Pelagic provides these standalone RM Agents through its SOA to act as standalone publishers, as well as the FDA-based publishers.

1.8.2. Data Listening

Pelagic Loggers publish not only the FDA data that they collect from applications and RM Agents, but also publish the FDA Data Schema themselves (because FDA Data Schema are data, that is, metadata). The lifespans of FDA data are controlled by the Data Writers' and Data Readers' QoS (e.g., HISTORY and LIFESPAN). The lifespans of FDA Data Schemas are controlled by the Logger's Data Writer, which locks the lifespan of an FDA Data Schema to match that of the Logger itself (using the Resource Acquisition is Initialization pattern).

With this publication technique, DDS subscribers can subscribe to both FDA Schema Topics and FDA Value Topics. Moreover, Pelagic provides pre-built C++ bindings to the streaming infrastructure as a value-added layer to DDS, referred to as an "FDA Listener". An FDA Listener is a C++ DDS-subscriber API that allows clients to discover FDA Schema that are currently available, to receive FDA Value streams, and to tie FDA Listeners to other tools, such as MATLAB, Complex Event Processing Engines, and any processing code that uses the C++ standard iostream and fstream interfaces.

Interest Tags. While a system is running, there are periods of time when an operator's interest may peak, for example, due to a sudden increase in the number of detections. As part of its SOA products, Pelagic provides a service, called "Interest Tags", that will insert user-defined markers into the time stream to identify those periods of operator interest. An Interest Tag is simply a pre-defined FDA that has a user-specified string and an on/off flag that is recorded when the Interest Tag Service is called. This service records its FDA data along with the rest of the FDA application data, so that when post-run analyses and playback are run, the analyst or operator can use any selected Interest Tags to filter the run data or skip to a point in time of interest.

DDS Agents. Pelagic also provides a “DDS Agent” in its SOA. A DDS Agent is an embeddable (API) or a standalone product that attaches to a specified Domain and allows its users to have it connect to and stream-in any Topics (not just FDA Topics). DDS Agents allow users to discover and listen to any Topics from any Domain. Similar to FDA Listeners, DDS Agents expose the C++ standard iostream and fstream interfaces to allow users to tap into data from arbitrary applications. For example, one use of a DDS Agent is to allow a legacy application to use its own data collection infrastructure to record data streamed-in from newer applications into its own data sets (i.e., de facto backward compatibility of DDS-enabled applications).

1.8.3. Application Control Framework

While the DDS publish-subscribe (asynchronous) infrastructure is effective at the monitoring aspects of system management, publish-subscribe is not the best interaction choice for the command and control of applications. Typically, a synchronous (blocking) interface is preferred for controlling applications, where a remote procedure call on an application will block the calling thread until a return status is received from the application. Such a blocking call protects the transactional integrity of the request and allows system control to proceed in a deterministic way. Effective overall monitoring and control of a system almost always entails a mix of synchronous command and control using point-to-point communications between known pairs and asynchronous notifications from unknown[6] publishers to unknown subscribers.

Pelagic provides a set of C++ framework classes that facilitate the staging of remote procedure calls from FDA Listeners (or any C++ programs). These framework classes encapsulate the

[6] Publishers can choose to make themselves known to subscribers if they include self-identification data in the data they publish, but the default situation is that subscribers do not know from where the data came.

underlying remote procedure technology (e.g., CORBA) with gateway lifecycle control and interface implementations for both client-side requests and server-side implementations.

An example use of the Application Control Framework is to build a client monitoring and control application that embeds a remote procedure call mechanism, importing the tracking control interface into the radar application that produces track data. Additionally, in this client application, embed an FDA Listener that subscribes to the rate of track creations as defined by the FDA. Furthermore, this application provides a presentation layer to show the user this performance metric. While a run is proceeding, the track creation rate may suddenly increase and the user decides to adjust the configuration settings on the application using the remote procedure interface using the embedded remote procedure call mechanism.

1.8.4. Playback Services

Once a system has run for a period of time and has logged its FDA data, Pelagic provides services for playing back any selected sets of FDAs from the log files for any selected duration of time. As illustrated in Figure 2, this data is formatted into its native streaming format and passed to a Streamer. Since the Streamer is agnostic to its buffering sources, it streams the data just as if it were originating in running applications, pushing it through its normal publishing channels. However, in playback mode, controls are available similar to those found on DVR players: stop, pause, fast-forward, fast-reverse. These capabilities are useful for reviewing previous system runs, for testing different configurations of data-consuming applications, for supplanting simulation drivers, and for performing experiments on Pelagic extensions (see below).

1.8.4.1. Complex Event Processing (CEP)

An <u>event</u> is defined as something that happens at a point in time. In the context of Pelagic operation, FDA publication can be viewed as a simple event. Figure 6 illustrates an example where the onboard monitoring system of a car may subscribe to data from several different sensors: a sensor that indicates whether the driver is in the seat and a sensor that indicates the current speed. Each sensor publishes the states of these sensors periodically. An FDA Listener in the onboard computer might observe that the driver has left the seat and the speed is above 10 mph. If the difference in time between these two readings is less than 2 seconds, then the onboard car monitoring system should autonomously notify its accident response service of a runaway car. In this example, the runaway car notification is a Complex Event, that is, a higher level event that is inferred from the occurrences of lower level events. Complex Event Processing performs operations on complex events, including reading, creating, transforming, and making inferences on them.

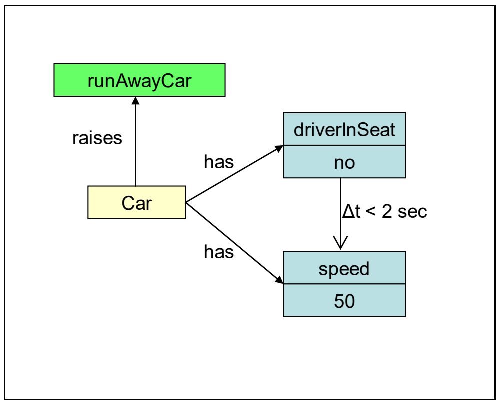


Figure 15. Illustration of Complex Event

Complex Event Modeler. Pelagic provides a CEP modeling tool, CEP IDE, that is layered on top of the FDA IDE. This CEP IDE enables users to browse a MetaLog for FDAs, which serve as low-level events, and display them as entities in a graphical composition window. The CEP IDE provides tools for specifying various types of event relationships, including timing, causality, and aggregation, and using them to link entities in the composition window, similar to a class diagram editing tool. The specification of conditions on the events and relationships that trigger a specific complex event is referred to as a trigger rule for the complex event. These trigger rules are expressed in an “IF-THEN” grammar predicated on simple boolean expressions on the properties of the participating events and their relations.

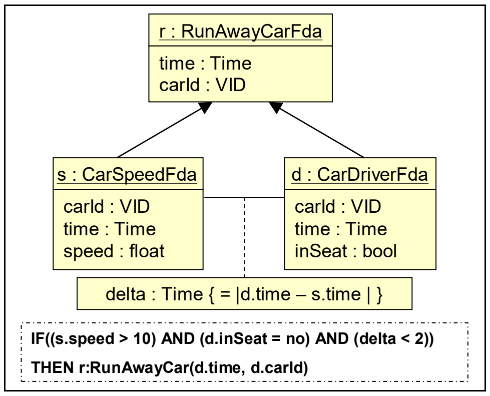


Figure 16. Illustration of Complex Event Modeling

Note that because the trigger rules are specified in the language of the application domain as opposed to being specified in a CEP-specific language, users and system engineers can readily understand the CEP trigger rules without needing to learn the language of a specific CEP tool.

CEP Inference Engine. The CEP Inference Engine subscribes to the FDAs that can serve as events. As FDA values are received, the CEP Engine determines what “IF” clauses are triggered, performs the execution of the inference rule to create the Complex Event, adds that Complex Event to its assertion set, and publishes the FDAs associated with the “THEN” part of the inference rule.

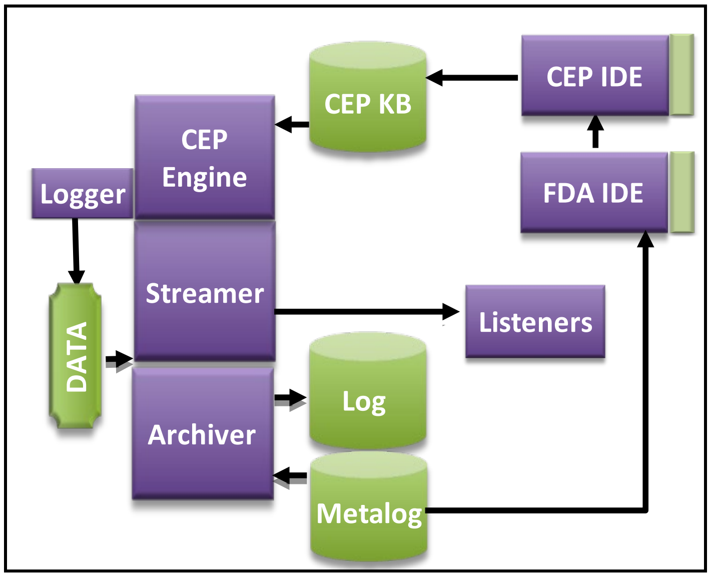


Figure 17. Complex Event Processing

1.8.4.2. Client-Defined Algorithm Download

Another form of pre-publication processing that is available with Pelagic allows users to download algorithms to a staging area of a publisher to be applied by the publisher to a data stream prior to publication of the data stream and, instead of (or in addition to) publishing the source data, publish the output of the algorithm. This capability assumes that the algorithm can be expressed as a script file for an analysis package that is available to both the subscriber and the publisher platforms, for example MATLAB. The following sequence of steps is suggestive. Variations in order and detail are likely. For example, in one scenario, the user may have an algorithm already defined and wants to inject it into an FDA stream to divert the stream to an output file. In another scenario, the user may find that the volume of raw data from some set of FDAs is unwieldy and wants to distill the raw data into a more succinct, more understandable set of processed FDAs.

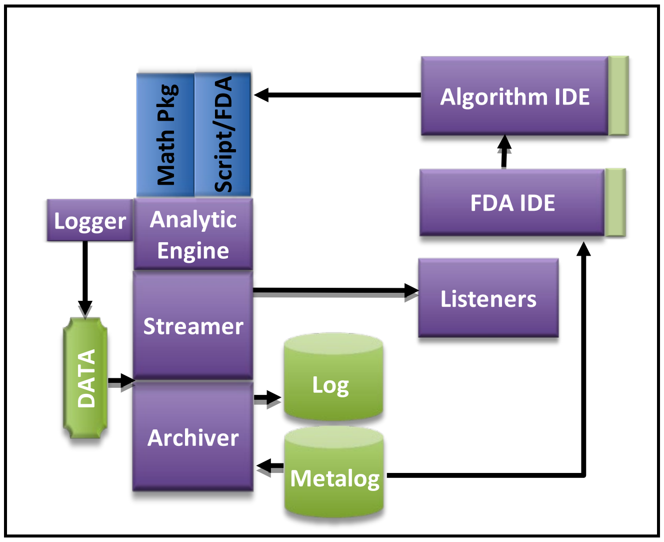


Figure 18. User-Defined Algorithms

Algorithm Definition. A user creates a data processing script using an analysis package and saves the script as a named file.

Algorithm Injection Specification. The user identifies which FDAs are to be earmarked for algorithmic processing. There are various ways that a user can find FDAs:

- If the user is currently listening to a running system, the FDAs are already being streamed to the client.
- Using a standalone FDA Listener, the user can determine what applications are currently publishing the identified FDAs by searching the current set of published FDA Schema Topics and then subscribing to the FDAs of interest.
- The user can search the Archived Logs for FDAs that have been published over some past period of time.
- The user can use the FDA IDE interface to a MetaLog to find the FDAs of interest.

Once the FDAs have been identified, the user decides which application runs should inject the algorithm. For current runs and Archived Logs, the user obtains the "runId" keys[7] of the published FDAs to indicate which application runs are to inject the algorithm. For future runs, the timestamp of the runId is left blank to indicate that all future runs should inject the algorithm (depending upon the injection-enabling settings in their Logger's configuration files).

FDA Bindings. Next, the user browses the MetaLog using the FDA IDE and finds the FDA that is to be bound to the input arguments of the data processing script. The user identifies the fields of the FDA that will be bound as input parameters to the algorithm. The user can choose to bind the results that are output from running the script to FDAs (existing or new FDAs) in addition to or instead of writing an output file. An XML file is created that identifies the script file, the output file (if any), the FDA input/output bindings, and additional processing settings (such as data ranges, time slices). By convention, the name and extension of this file will be used by Loggers to obtain the find the injection request.

Algorithm Submission. The user downloads the script and bindings files to the server file system area designated for Logger configuration files. The user may also modify Logger startup configuration files to enable algorithm injection processing and/or to name the particular bindings file just submitted.

Algorithm Execution. At application program startup, the Logger reads its startup configuration file. This file indicates if the Logger is enabled for injecting client-supplied algorithms and what client-supplied algorithms to accept. The latter setting is a list of named bindings files, with wildcarding allowed. If enabled, the Logger reads the list of bindings files and, for each bindings

[7] Note that the "runId" key contains two fields: a logical field that identifies the application's executable program, and a timestamp field that identifies the time at which a run started.

file, checks whether all the identified FDAs are enabled for streaming, uses the mapping specified in the bindings file to set up the data marshals for each input argument and each output value, and injects the algorithm into its publication stream using a chain-of-command control pattern. The Logger can be configured to execute the algorithms directly in the Logger's thread or in their own thread. As an FDA value is streamed, if it is needed in the chain of algorithms, its participating fields are passed as arguments to the algorithm for processing by the algorithm. After the algorithm is complete it adds the results to the output file (if so configured) and/or publishes the generated FDAs (if so configured).

1.8.5. Ontology-Based Information Exchange Services

FishEye plans[8] to augment Pelagic with Ontology-Based Information Exchange Services (IES) that are based on domain semantics using information exchange standards, such as UCore[9] (XML) and C2 Core[10] (OWL) using the FDA infrastructure as a base.

[8] FishEye has submitted a Small Business Innovation Research proposal to seed the planned work.

[9] UCore (Universal Core) is an information exchange specification and implementation profile developed by a collaborative of the Departments of Justice, Homeland Security, Defense, and National Intelligence. It is based on a vocabulary of most commonly exchanged concepts (Who, What, When, Where) and an XML representation of those concepts. It includes extension rules to allow tailoring to specific mission areas. UCore supplies a messaging framework to package and unpackage the content consistently

[10] C2 Core (Command and Control Core) is an extension of UCore for the Joint C2 Community of Interest that provides a rich ontology for the C2 domain, sufficient for interoperability on the Semantic Web.

1.8.5.1. Ontolog

In computer science and information science, an ontology is a formal representation of what is known about a domain expressed as a set of concepts and the relationships between those concepts. In comparison, an FDA in Pelagic is a formal representation of what data is available from a running system and how elements of the data contribute to an information concept of interest to a user, however, in the current version of Pelagic, that information concept is only knowable by ad-hoc connotation. The vocabulary of the data elements is whatever the system developers found useful for advancing a design and implementation and may not be a consensus representation of any given community of interest. Likewise, the naming and description of an FDA is an arbitrary choice by a user and, therefore, can inhibit community-wide recognition of the intended meaning and usefulness of an FDA. The planned ontology extensions will make FDAs much more effective across a community of users by organizing them into an ontology agreed upon by its users. To this end, it is planned to augment the Pelagic tool to model and capture ontologies, to maintain them in an Ontolog, and to allow users to name and organize their FDAs within the concepts and relationships of those ontologies.

A preliminary top-level design has been done as a first step toward this goal, shown in Figure 19. Each domain will have its own ontology, so a Domain class will be introduced so that an object of Domain will serve as a root to each ontology. Domains can have sub-domains, which is modeled by a simple “subdomain/superdomain” association. Within a Domain, the concepts will be represented by DomainEntity classes, and the relationships between concepts will be represented by Relation classes. Detailed characteristics of how DomainEntities interact are modeled by the Role association class. The data that evaluates the state/condition of a concept at a given time and space, by definition, is an instance of an FdaSchema.

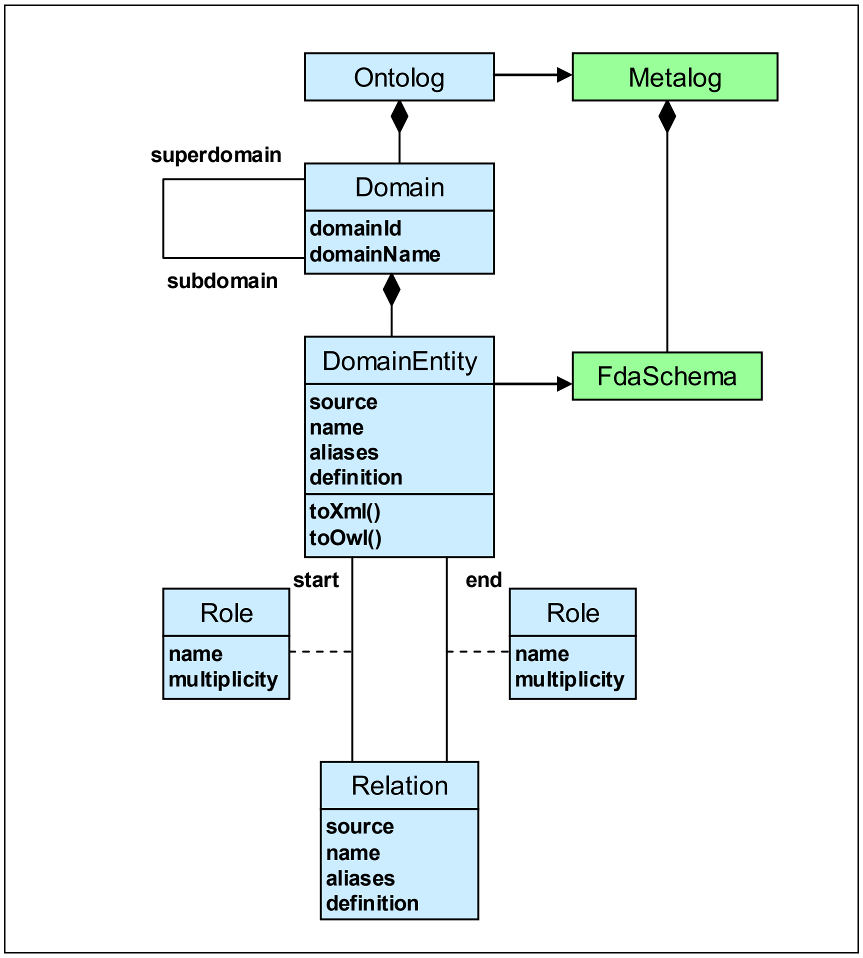


Figure 19. Preliminary Design of Ontolog

1.8.5.2. Ontology-based Data Distribution Services

With this modeling approach, a user of the ONTO IDE can specify an ontology as a graph with DomainEntity objects as nodes and Relation objects as edges and then link DomainEntity objects to FDAs to provide domain-specific views of the FDA data distribution service. As a result, subscribers can opt to register for information about domain concepts (as an alternative to registering for FDAs) and receive publications in terms of the concepts-and-relationships of their COI. Additionally, in the spirit of the Semantic Web, Pelagic will be enhanced with export interfaces to publish information in both XML and OWL formats.

[i] Revision to paper of March 1, 2017

[ii] Pelagic and the Pelagic Real-Time Platform are Trademarks of FishEye Software, Inc. Pelagic capabilities are protected under US Patent 9652312 and contain SBIR Data Rights.

MetaGen, MetaLog and MetaData Injection are Trademarks of FishEye Software, Inc.

[iii] Pelagic™ is a rebranded from the prior name of the Real-Time Tool Kit™ (RTTK™)