****The Framework for Designing

Cloud Computing

Applications for Distributed Big Data Reconstruction and

Analyses

# Background and Motivation

The principal goal of CLARA is to provide a computing environment for efficient data processing. Under the data processing efficiency we assume two basic concepts:

• Data processing performance optimization

• Data processing application agility

## Data Processing Performance

Today, big data is generated from scientific (for e.g. high energy and nuclear physics experiments, earth science observatories, cosmology, bioinformatics, etc.) as well as commercial sources (for e.g. social networks, digital imaging, search engines, etc.). Data processing communities are faced with a challenge to filter, aggregate, correlate, analyze and report results of large volumes of data. Besides data volume, data throughputs and raw data production speeds, raw data varieties and heterogeneities are other challenges of big data. Although big data batch processing is the most acceptable and used solution of the problem, there is an increasing demand for non-batch processing on big data that are capable of providing substantially faster and near real-time data processing capabilities. CLARA design focus is on two new traits: real-time data processing and stream processing solutions. Data driven and data centric architecture of CLARA presents solutions, capable of processing large volumes of data interactively and substantially faster than batch systems. Data processing performance increase is largely due to big data stream processing, solving a problem that input data must be processed without being fiscally stored. CLARA event-processing assumes that data is presented as set of identifiable events, and if so, CLARA services can be used to develop high performance complex event processing engines that are capable of detecting patterns of activity from continuously streaming data.

## Data Processing Application Design

To achieve quality data processing of distributed big data an intellectual input from diverse groups within large collaborations must be brought together. Data processing in a collaborative environment has historically involved a computing model based on developing self-contained, monolithic software applications running in batch-processing mode. This model, if not organized properly, can be inefficient in terms of deployment, maintenance, exception handling, update propagation, scalability and fault-tolerance. Also the rate at which computing hardware technologies are advancing creates additional challenges for legacy software applications that must adopt to satisfy ever-growing data processing requirements. Legacy software adoption process to new hardware technologies is quit painful, resulting in code fragmentations and ad-hoc extensions. This has led to computing systems so complex and intertwined that the programs have become difficult to maintain and extend.

Software technologies are evolving as well. There are many new software architectures and improved high level programming languages in the market. Software application development is a process of writing, testing, debugging, and maintaining the source code of particular algorithms. The source code is usually written in one of the high level (HL) programming languages (such as Java, C++, Python, etc.). The process of designing a software application requires expertise in many different subjects including knowledge of the application domain, specialized algorithms, formal logic, and of course, syntactic and semantic knowledge of the chosen HL programming language. This is a description of a widely adopted traditional approach that is used to design and develop software applications. So, what if a user is an expert in a specific application domain yet has a limited knowledge and experience in software programming. Obviously this user cannot actively contribute to the process of developing domain specific software applications. However, the same user can very effectively design an application using available software application building blocks: a process that does not require writing and/or compiling a source code. Data processing applications have a very long time (reprocessing a data over and over again), and the ability to upgrade technologies is therefore essential. Data processing applications must be organized in a way that easily permits the discarding of aged software components and the inclusion of new ones without having to redesign entire software packages at each change. The addition of new modules and removal of unsatisfactory ones is a natural process of a software application evolution over time. Experience shows that software evolution and diversification is important and results in more efficient and robust data processing applications.

Software applications designed within the CLARA framework differ from a traditional approach in two major ways. First, software application is composed of interlocking software bricks called services. Services are linked to each other by the data link and are hiding technology (for e.g. HL programming language) and algorithmic solutions used to process data. A CLARA service has inputs, is capable of processing data, and producing the output data. Once a given service receives valid data, that service executes its engine, produces output data, and passes that data to the next service in the dataflow path. The second and critical difference is that CLARA applications execute according to the rules of data flow instead of a more traditional programming approach, where sequential series of instructions (lines of code) are written to perform a required algorithm. In this respect CLARA application design promotes data and data flow as the main concept behind any data processing algorithm. CLARA service execution is data-driven and data-dependent. The flow of data between services in the application determines the execution order of services within the application. These differences may seem minor at first, but the impact is revolutionary because it allows the data paths between application building blocks (bricks) to be the application designer’s main focus. As a result a CLARA application is more robust and agile, since application building blocks can be improved individually and be replaced. This type of application is more fault-tolerant, since faulty block can be replaced without bringing down entire application. CLARA application is elastic, since at run-time new data processing services can be added to enhance application functionality or add/remove services to fit available hardware computing resources.

# The Framework

## Programming paradigm

The CLARA framework uses a service-oriented architecture (SOA) to enhance the efficiency, agility, and productivity of data processing activities. Services are the primary means through which data processing logic is implemented. Data processing applications, developed using the CLARA framework, consist of services, running in a context that is agnostic to the global data processing application logic. Services are loosely coupled and can participate in multiple algorithmic compositions. Legacy processes or applications can be presented as services and integrated into a data processing application. Simple services can be linked together and presented as one, complex, composite service. This framework provides a federation of services, so that service-based data processing applications can be united while maintaining their individual autonomy and self-governance. It is important to mention that CLARA makes a clear separation between the service programmer and the data processing application designer. An application designer can be productive by designing and composing data processing applications using available, efficiently and professionally written software services without knowing service programming technical details. Services usually are long-lived and are maintained and operated by their owners on distributed CLARA service containers. This approach provides an application designer the ability to modify data processing applications by incorporating different services in order to find optimal operational conditions, thus demonstrating the overall agility of the CLARA framework.

## Services and service interactions

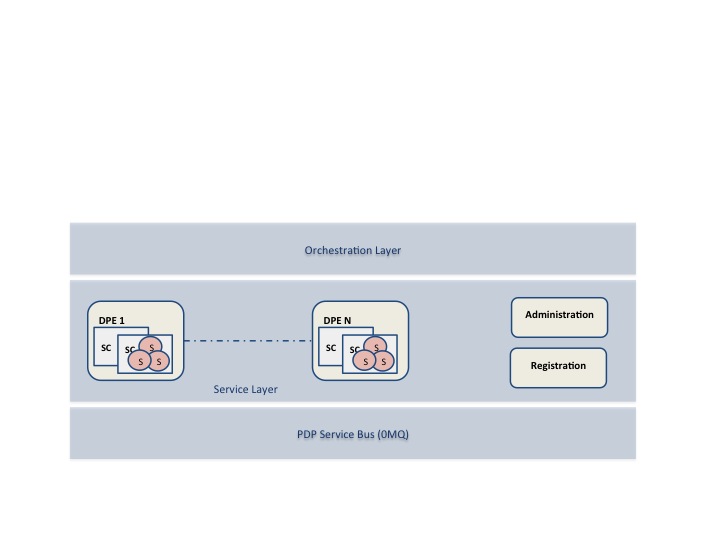
Message passing is the most popular communication model for distributed computing. It is key for building SOA-based frameworks. This model is attractive due to the fact that messaging does not emulate the syntax of programming language function calls (like CORBA and RPC for example). Instead, structured data messages are passed between distributed components (i.e. services). In this distributed communication model success largely depends on the clever design of the message structure: a communication envelope that describes not only transferred data but also communication and service operational details. In order for a service communication to be truly useful, every party has to share/use the same vocabulary for expressing the communication details (i.e. common message-interface).

The CLARA framework provides developers with the means for interacting with services based on the publish-subscribe message exchanges. But such explicit interactions, where a service invokes operations exported by the predefined interface of a well-known target service, are only one piece of the messaging puzzle. To make this clear, consider a persistency service that converts CLARA transient data into a ROOT tree. Using CLARA tools one can link for example a charge particle tracking service to a ROOT persistency service for storing reconstruction results in a ROOT format. In this particular scenario, the persistency service (i.e. invocation target) is known in advance and the responsibilities between the requestor service and the provider service are defined in a service contract. But that same messaging strategy is far less suitable for indicating *event occurrences,* for example a input-out exception detected by the persistency service. In such situations, the developer of the persistency service either doesn’t *know* who is interested in the event, or doesn’t want to hardcode the event handling logic in the service. Indeed, doing so would increase its complexity and reduce its reusability and maintainability. What CLARA provides for such cases is a way to deliver event notification to services that register their interest in one or more events. This is possible due to the CLARA message envelope design (service communication message structure) that contains event notification.

CLARA services are loosely coupled, since there are no dependencies between services because event-producing services typically invoke generic operations such as execute/notify (rather than target service specific algorithmic methods). Even more, a service developer is unable to predict future customers (i.e. services that will be linked to it). Only a final physics data processing application (service composition) designer knows the event/data flow outline. Rather than contacting services directly, the *implicit* invocation mechanism only signals that output-data is ready (an event has occurred) and it does not say what needs to be done to that data (how to react to that event). This clearly improves its maintainability, and it simplifies data processing application-reengineering processes. CLARA services can be considered as event handlers for one another. Since event handlers are external to other services, the workflow modification of a handler does not require modification of any event producing services.

## Design architecture

This framework was designed based on a specific set of principles. The fundamental unit of CLARA based data processing application logic is the service. Services exist as independent software programs with a common interface defined by the framework. User classes, encapsulating specific algorithms and compliant to the required interface, can be presented as CLARA services (the CLARA Software-as-a-Service: SaaS implementation). Each service has its own set of data processing functionalities. These functionalities or capabilities, suitable for invocation by other services, can be discovered via registration information available from the CLARA platform registry services. One of the service design recommendations is to keep a small and simple service code base, which will help future programmers to easily extend, modify, maintain and port services. Services must be agnostic to any eternal data processing logic. Services must be discoverable and able to take part in complex service compositions. By standardizing communication between services, adapting a data processing application to changes in one of its components becomes easier and simplifies data transfer security (for example by deploying a specialized access control service). The CLARA architecture consists of tree layers.



The first layer is a service bus that provides an abstraction of the 0MQ publish-subscribe messaging system. Every service or component from the orchestration layer communicates via this bus, which acts as a messaging tunnel between services. Such an approach has the advantage of reducing the number of point-to-point connections between services required to allow services to communicate in the distributed CLARA cloud. 0MQ is a messaging system that scales perfectly to tens of thousands of processes, running over many boxes with many cores as wanted. This simple API based on BSD sockets implements real messaging patterns like topic pub-sub, workload distribution, and request-response. It is design as a library that is linked with CLARA framework. 0MQ is licensed as LGPL code and can be used with no licensing issues in closed-source as well as free and open source applications. The service layer houses the inventory of simple/entity and complex/composite services (linked service chains presented as a single service) used to build data processing applications. An administrative registration service stores information about every registered service in the service layer, including address, description and operational details. The orchestration of data analyses applications is accomplished by the help of an application controller, resident in the orchestration layer of the CLARA architecture.

### Service categories

CLARA specifies four types of services: entity, utility, task and orchestrated task.

* Entity services are highly reusable and generic. They are atomic enough to take part in different service compositions.
* Users find many self-contained and legacy software systems very useful. These systems can be presented as utility services. The difference between entity and utility service is size and complexity. We hope in the future that the utility service definition will be deprecated. Currently the legacy software applications temporarily are labeled as utility services before they will be categorized (after proper segmentation and modularization) as entity services.
* Task and orchestrated task services are both composite services, with the only difference being that task-services are self-governed, while orchestrated services are aggregated services controlled by the software components from the orchestration layer of the framework.

### Service compositions

A service composition is comprised of services that have been assembled to provide the functionality required to accomplish a specific data processing task. CLARA distinguishes between two types of service compositions: primitive and complex. Primitive compositions use message exchange across two or more services. Complex compositions, however, require an orchestrator. Because the frameworks requirement for services is to be agnostic to any physics data processing logic, one service may be invoked by multiple data processing applications, each of which can involve that same service in a different composition. A collection of entity services can form the basis of a CLARA service repository that can be independently administered within its own physical deployment environment. So, the CLARA framework helps to build services, service compositions, and service inventories. The service-oriented approach of CLARA changes the overall complexion of a data processing application. Because the majority of services delivered are reusable resources agnostic to analysis, they do not belong to any one application. By dissolving boundaries between applications, the data processing is increasingly represented by a growing body of services that exist within a continuously expanding service inventory.

## 

## Service engine

CLARA implements SOA SaaS as a way of delivering on-demand, ready-made data processing solutions (“service engines” in CLARA terminology) as CLARA services. The CLARA data processing application user uses a service, but does not control the operating system, hardware or network infrastructure on which they are running. The quality of the data-processing application (including syntactic, semantic qualities and performance) depends highly on the quality of constituent services. It is, therefore, absolutely critical to test and validate an engine before deploying it as a CLARA service. Physics data-processing engines must be validated with respect to workflow, thread-safety, integrity, reliability, scalability, availability, accuracy, testability and portability.

### Data processing environment, service container, and SaaS implementation

The highly distributed nature of CLARA is largely due to traits of the CLARA service container. A service container is the physical manifestation of an abstract service representation and provides the implementation of the CLARA service interface. Service container is a way of logically grouping services. A service container runs within the CLARA Data Processing Environment (DPE) that provides a complete run-time environment for service execution and operation. DPE presents a shared memory that used by service containers to communicate transient data between services within the same DPE. This prevents unnecessary copying of the data during service communications. Services in a DPE are group in multiple service containers. The CLARA service container allows the selective deployment of services exactly when and where you need them. In its simplest state, a service container is an operating system process that can be managed by the CLARA framework. A service container is capable of managing multiple instances of user service engines. Several service containers can coexist within the same DPE providing the logical grouping of services. Service containers may also be distributed across multiple machines for the purposes of scaling up to handle increased data volume. CLARA administrative services start service containers in a specified DPE. They also monitor and track functionality of service containers by subscribing to specific events from a service container, reporting the number of requests to a specific container, as well as notifying when a successful execution of a particular service (or its failure) has occurred.

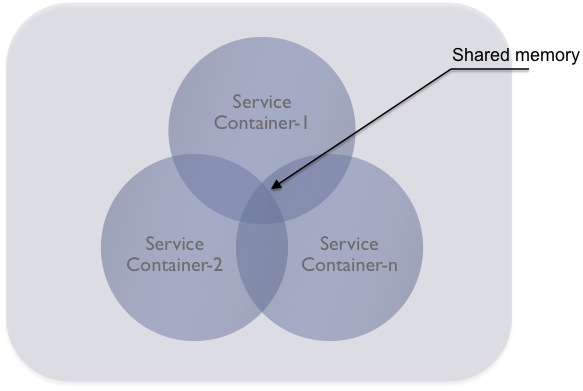


Figure. CLARA data processing environment houses multiple service containers. Service containers use DPE shared memory for transferring data between services within the local (DPE) environment.

## 

## Service registration and discovery

The core of the CLARA registration and discovery mechanism is the normative registry service that the CLARA services and service containers are registering with. The normative service, which is started within the DPE, functions as a naming and directory service for entire CLARA cloud infrastructure. Services and service-containers in the CLARA registry are described using unique names, types and descriptions. The CLARA naming convention defines the service container name as:

***DPE\_host\_IP\_address:service\_container\_name***

where the *service\_container\_name* is an arbitrary string specified by the user. Likewise, the service name is constructed as:

***DPE\_host\_IP\_address:service\_container\_name:service\_engine\_name***

The engine name is the class that implements CLARA interface. Querying the name and a service description defines the service discovery process. The service is advertised by its service description in the registry. By retrieving this service information, the user can discover services. Note that at the moment the service and/or service container discovery process is modest, and is not taking into account service functional information.

## Service granularity

Service granularity describes the amount of data processing performed by a single request to a service. There is no single suggested size for all CLARA services. To define the size of a service one should take into account the following (application specific) design requirements:

* Service invocation/request frequencies
* Service network distribution
* The data amount passed during the service interaction

In addition to the distribution and data transfer, it is important that the granularity of a service match the functional modularity of a data processing application. One should also consider designing services with finer granularity in case there is a functionality that is going to be cloned and/or changed over time.

## Service accessibility

Service accessibility describes the intended class of users of a service. CLARA implements two types of service visibility, described as either public or private. Public visibility means that all users within the CLARA cloud infrastructure are able to discover and use the service. Private means that service can be discovered, but will respond to specific clients (orchestrators and/or services) only.

## Service invocation

CLARA services are invoked using SOA most common Request/Reply communication mechanism. Two separate implementations of this mechanism are supported: synchronous and asynchronous. The basic mechanism of the synchronous service communication is when the requestor service sends a request to a service and waits. When the service has processed the request, it sends a reply. The requestor receives the reply and resumes it’s internal processing.

The asynchronous communication mechanism is based on an event-based approach, most commonly known as publish/subscribe communication. This type of communication is native to CLARA, which is using 0MQ publish/subscribe middleware for message passing between services. In this mode, a requester defines an event or subject of interest and *subscribes* to this event. Next requester sends its request to the receiver service along with the subject to which the response must be returned. Whenever the service is ready it *publishes* the response to the requested subject.

## Transient data envelope

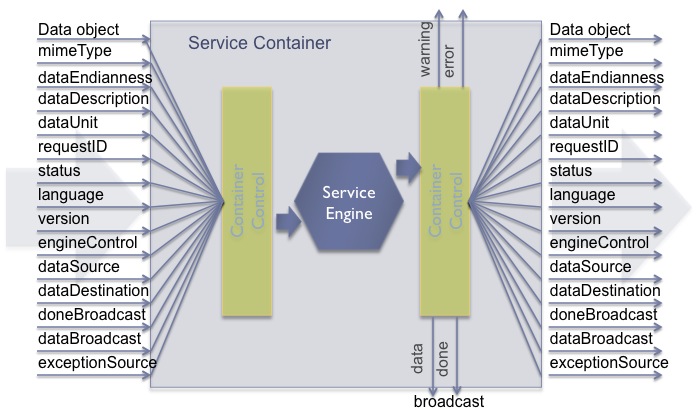
Think of a CLARA service as a combination of its interface (the public view of the service), and its algorithmic implementation (the private view of the service). A CLARA service interface provides the following functionalities:

* Hides the details of the implementation
* Expresses the service’s functions
* Provides parameters for the service operations

A CLARA service is a software component that offers functionality on a semantic level by specifying its interface in a standardized way. A semantic level refers to a service that is self-descriptive in a way that it can be consumed dynamically and loosely coupled by other CLARA services with a consistent understanding of communicating data. The major backbone of the CLARA system is data. Data fed to services, generate a data processing action. All data sent between services are required to be self-descriptive.

A transient data envelope that contains service data is the main object passed between CLARA services. The mutual understanding and acceptance of this object couples services. When we say CLARA services are loosely coupled we mean that this transient data object is the one and only physical coupling between CLARA services.

In the CLARA framework a special class that has implementations both in Java, C++ and Python represent the transient data envelope. This class contains fields and methods to pack and retrieve transient data as well as describe data, service communication and service operational details. Fields of the envelope are show in the Figure below.



Figurw. Transient data envelope and a service container with a single service engine

The meta-data segment of the envelope defines the type (mime-type), description, and measurement unit of the transient data object. The communication segment of the envelope is designed to inform the receiving service of the high level programming *language*, the *version* of the engine, and the engine execution *status* of the source service. Even though we consider only event level parallelization for CLARA based data processing applications, we do not discard the possibility of having multi-tier services (services that are shared by multiple service based applications) in service compositions for building certain data processing applications. The *requestID* is designed to synchronize request/response pairs.   
The control segment of the envelope informs the receiver the name of the service responsible for creating the data stored in the envelope (*dataSource*), specific service name that the data is addressed to (*dataDestination*), as well as the name of the service that threw an exception during engine execution (*exceptionSource*). This segment of the transient envelope is used for controlling data flow between services of an application. Even though service engines can define additional data links, these control fields of the CLARA transient envelope are designed to control the data flow within application-defined service link diagram. Any exception thrown during the execution of the service engine will be passed on following the predefined data pass of the data processing application, yet the control segment *exceptionDestination* field will allow for specifying the exception data path in every communication. Fields *doneBroadcast* and *dataBroadcast* can be set by an external orchestrator of an application or by any service engine. These fields are used to store the name of a service. This will notify a service container to inform the completion of a particular service engine execution. The done broadcast message envelope is designed not to have the data object in it. Contrary to the fact that user can request done and data to be broadcasted, errors and warnings are broadcasted by the service container in case service engine or service container detect a specific alarm condition. It is important to mention that the name of the service (canonical name) defines the physical location of the service. The location information is important to design data processing applications with the location-optimized communications within the CLARA network distributed cloud. The location information is also useful when querying sets of data generated in an area of interest (for example any orchestrator that subscribes data from a specific service). Data versioning is a relatively new requirement within data processing, yet very useful for reporting purposes. It is a common means to track services that processed data. This is useful within the system because of how service data processing algorithms and solutions are hidden from direct access. The *engineControl* of the control segment of the transient data envelope is designed to control/configure (for example to alter the configuration of the service specified for communication) service engine at the run time.

## Service communication monitoring

Auditing and logging play an important role within the distributed CLARA cloud. The anticipated complexity of data processing applications, scaled over multiple CLARA data processing (DPE) environments and multiple service containers, requires tracking and constant monitoring of service communications and in some cases data flows between services. Reliable service communications ensure that the data gets to its intended destination, thus assuring overall PDP application quality. As part of the framework’s administrative and management capabilities, CLARA provides auditing and logging services. These services are deployed within the CLARA DPE and can have multiple means for tracking service communications and data. System-level information about the health of the service itself and the flow of messages can be tracked and monitored. PDP application-level auditing, logging, and fault handling are accomplished through the transient data envelope metadata fields, namely the service execution status and the data description. The framework uses service data endpoints to deliver system-level errors, such as service engine thrown exceptions, as well as application-level errors.

### Exception propagation and reporting

There is an underlying philosophy behind the way that the communication tracking, system errors, and application faults are handled. In addition to the normal handling of the outgoing flow of transient data, additional destinations are available to the service for auditing the message and for reporting errors. The service container implementation uses special message subjects for reporting/tracking, system errors and application fault events (see paragraph titled “Transient data envelope”). Anyone interested in these events can subscribe to the specific message subject and receive notification on the occurrence of specific events. From the service implementation's point of view, in the case of an exception it simply creates a CLARA transient data object with proper description of the event and publishes it to a specific, predefined message subject. The CLARA framework takes care of managing processes, such as auditing, logging, and error reporting to all interested (subscribing) services and/or service orchestrators. This approach provides a separation between the implementation of the service and the details surrounding fault handling. The implementer of a service need only be concerned that the service has a place to put such information, whether it is information concerning the successful processing of a good data, or the reporting of errors and bad data.  
Exception events can be handled at both the individual service level and the service orchestrator level. A data processing application may make use of different implementations of individual services over time. The tracking of a fault occurrence or the auditing of an individual message can be tied to the context of a data processing application’s independent orchestrator that overlooks the entire cloud deployment exception status. For this purpose the CLARA framework provides a normative service that subscribes to specific exception events and logs them in the CLARA database.

## Cloud formation

Conceptually a CLARA data processing application designer and/or user acquires data processing services from a CLARA network distributed environment (i.e. cloud) and then designs and runs an application based on selected services. Therefore, CLARA cloud offers users services to access PDP algorithms and applications, persistent and/or transient data resources. Figure below shows the relationship between services and the data transfer modes between services in a CLARA cloud environment. A CLARA cloud consists of multiple data processing environments (see paragraph “Data processing environment, service container, and SaaS implementation”) each providing a complete run time environment for service deployment and operation. Each of the DPEs of the CLARA cloud host at least one service container with at least one service.

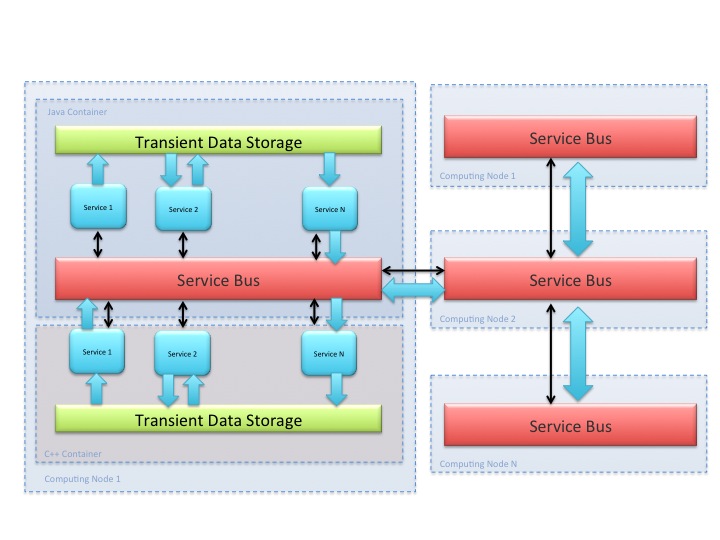


Figure. CLARA Cloud formation

Scalability and flexibility are the most important features driving the emergence of Cloud computing. CLARA services and DPEs can be scaled across geographical locations, software configurations and performances. For data transfer efficiency reasons, transient data communication between the same language service containers, within a DPE, is established through shared memory. The data that is sent across language barriers or across the network is transferred through pub-sub middleware.

# CLARA SaaS In Nutshell

The CLARA service model assumes that the software, as well as the solution itself is provided as a complete service. This approach is referred to as Software as a Service (SaaS). A CLARA service may be concisely described as a software application (service engine) that is deployed on a CLARA DPE and can be accessed locally as well as globally over the Internet. With the exception of a user’s and other service interactions with a service engine, all the aspects of a service are abstracted away (including algorithmic solutions, composition, inheritance, technology, etc.). As was mentioned, CLARA SaaS supports multiple users and provides a shared data model through a single-instance, known as a multi-tenancy model (i.e. services that are shared between multiple data processing applications). So, the use of the multi-tenancy model in the CLARA SaaS implementation dictates the only requirement to service engine: the service engine must be thread enabled or thread safe.

## All you need to convert a software application into a CLARA service.

In order to present software as a CLARA service we need 2 things:

* Understanding of the CLARA transient data envelope
* Implementing of the CLARA service interface (or inherit from the CLARA service abstract class)

The CLARA transient data envelope is described in the previous chapter. The CLARA framework uses Google’s protocol buffers to provide the transient data representing classes in Java, C++ and Python. Protocol buffers are language-neutral, platform-neutral, extensible mechanism for serializing structured data. CLARA transient data is described using protocol buffer syntax and a source code is generated that allows easily write and read serialized transient data to and from services and using a variety of languages – Java, C++, or Python.

An object of this class is passed as a parameter to the CLARA service interface methods. The table below describes the CLARA service interface methods.

|  |  |
| --- | --- |
| Method signature | Description |
| public void configure (TData data); | Service configuration |
| public TData execute (TData data); | Service engine execution |
| public TData execute\_group (TData[] data); | Service engine execution |
| public String[] get\_states(); | Possible engine states |
| public String get\_current\_state(); | Service engine current state |
| public Enum get\_accepted\_data\_type() | Engine’s input data type |
| public Enum get\_returned\_data\_type() | Engine’s output data type |
| public String get\_author() | Returns name of engine author |
| public String get\_description(); | Functional description of engine |
| public String getVersion(); | Returns version of engine |
| public void dispose(); | Graceful withdrawal of engine |

Table. CLARA service interface. TData is the class that represents transient data object.

## CLARA transient data protocol buffer description.

The CLARA transient data is described using Google’s protocol buffer IDL (Interface Definition Language) and then pre-compiled into source code for the target languages (Python, Java and C++) of service engines. Below is the CLARA transient data protocol buffer description file.

option optimize\_for = SPEED;

message transient {

// ........... Defined by the engine .........

// The state of the service after execution

optional string state = 1;

// Overall status of the execution

required iType status = 2 [default = INFO];

// Actual payload. Note: always byte[]

required bytes data = 3;

// Data type

required dType dataType = 4;

// Data version

optional string dataVersion = 5;

// Data author, i.e. engine developer. Note: set by the engine or obtained

// by calling engine get\_author method.

optional string dataAuthor = 6;

// ........... Defined by the application orchestrator .........

// Used for communication synchronization

optional int32 request\_id = 7;

// The name of the service that presents the transient input data

required string sender = 8;

// If set to true, service container will broadcast exceptions to

// [exception\_<service\_name>, requestId, exceptionType <iType>,exceptionString <String> ]

optional bool exceptionMonitor = 9;

// If set to true, service container will broadcast resulting data to

// [ data\_<service\_name>, requestId, transient\_data ]

optional bool dataMonitor = 10;

// Reserved control field

optional cType controlR = 11;

// Clara service based application composition

// i.e. link schema (list of linked service names) e.g. s1+s2;s1+s3,s4,s5+&s6

optional string composition = 12;

// Action defines which of the CLARA interface methods must be called on engine.

required aType action =13;

//.............. Enumerations ...........

// Service requested actions.

// Actions are in reality the names of Clara interface methods (e.g. execute, config, etc.)

enum aType {

EXECUTE = 0;

CONFIGURE = 1;

}

// Service engine controls

enum cType {

SKIP = 0;

}

// Transient data type. Note that data is always passed as a byte[] and this enum indicates how to handle the data

enum dType {

STRING = 0;

JOBJECT = 1;

COBJECT = 2;

POBJECT = 3;

NETCDF = 4;

HDF = 5;

}

// Information severity types

enum iType {

ERROR = 0;

WARNING = 1;

INFO = 2;

}

}