Software Design for Distributed Sensing and Computing Tasks

Anthony Cowley, Hwa-Chow Hsu, Camillo J. Taylor
GRASP Laboratory
University of Pennsylvania, Philadelphia, PA, USA, 19104

ABSTRACT

In this paper we illustrate the benefits of a strongly typed software design framework, explore the difficulties in applying such a system in a distributed sensing setting, and describe the extended type system we have created and fielded in multi-robot sensing experiments. While the built-in type systems of modern object-oriented languages provide much of the functionality we desire, there is a level of synchronization required by both static and dynamic linking that limits the applicability of such a system to a scalable distributed sensing and computing platform. We show how the limitations of these robust strong type systems can be overcome, allowing one to bring their power to bear on distributed sensing. By adhering to a formal, well-supported type system, our framework offers a scalable approach to dynamic resource discovery and exploitation. A natural consequence of the platform’s design is a higher level design system for building multi-agent programs that itself enforces type safety when pairing data sources and sinks, both when the distributed task is being launched and as the task dynamically reconfigures itself to exploit new resources.

Keywords: distributed computing, strong-type system, code reuse

1. INTRODUCTION

The desire to field ever more complex configurations of sensors and computing platforms has brought with it the need for new development systems attuned to the particular needs of this problem domain\textsuperscript{1,2,3}. One demand of these projects is that multiple software developers be able to collaborate effectively and efficiently. Another is that the resultant software library be robust. This second demand may not be particular to the problem domain under consideration, but the need for teams of largely independent, research-focused developers and runtime operators makes it a harder ideal to realize than it would be under more constrained development and deployment environments. A final consideration is that traditional software development techniques were not designed to generate applications destined for distributed deployment or fluid reconfiguration. The notion of fluid deployment configurations suggests the further desire to generate software that is not only manually reconfigurable at runtime, but programmatically reconfigurable. This requirement implies that autonomous agents need to be able to perceive and reason about available software. Fortunately, these disparate design goals can all be realized by a strong-type system capable of efficiently supporting distributed execution environments. While alternative systems such as TinyOS\textsuperscript{4}, Sun’s Jini platform\textsuperscript{5}, and others\textsuperscript{6,7,8} address some of these concerns, these systems target different levels of hardware capability and have programming requirements that make them less than ideal for some applications. The system we describe has been designed to target networks of relatively capable nodes, require minimal developer effort to support, and increase reliability and usability by emphasizing type design.

2. TEAM DEVELOPMENT

The demands that team research places on a development system are defined by communication constraints. Specifically, as the number of people in a team rises to meet the demands of ever more complex systems, the cumulative cost of face-to-face interactions rises exponentially. While personal interaction is well-suited to certain situations due to its reactive, high-bandwidth nature, redundancy in this area, like many others, can cripple the team’s effectiveness. One area that typically consumes a large amount of time is the effort spent by developers to familiarize their colleagues with existing code. In particular, for the pieces of code generated by multiple developers to interact
properly, the interfaces must be well-understood by all involved. Furthermore, if the use of a given piece of code is understood, it can be reused and repurposed by developers other than the original author(s). This knowledge – essentially, what a piece of code does and how to make use of it – can be conveyed in a number of ways.

2.1 Interface Specification
The most obvious way of sharing knowledge about how to use a piece of code is to have the original developer explain his or her code to any who wish to use it or integrate it into a larger whole. The impracticality of this method over time, along with its inability to scale to larger team sizes, has prompted the software development community to develop and encourage alternative techniques of code annotation and contextualization. Historically, comments in source code files have been advocated strongly. While we make heavy use of structured commenting patterns in our work, it is also clear that this is not an ideal solution to the interface description problem in and of itself. Immediately evident is the problem that the lack of structure in plain text comments makes them difficult to consume both programmatically (in the case of autonomous agents discovering and reasoning about available code) and even manually, due to inconsistencies in what information is contained in comments, where that information can be found, and the syntax in which the information is presented. Structured interface description models (using XML or another structured syntax) attempt to make the information more easily consumed, but they do not address a second significant problem: the detachment between comments and executable code. That is to say, comments are still technically optional and often fall out of date. This is due to the fact that a developer iterates through many “code, compile, test” cycles over the course of development of each piece of a program, and the process of writing comments that describe the code being written is seen as an unwanted burden. The critical point of failure that we have identified is the compilation stage. When a developer updates his or her code, perhaps changing the way it must be interfaced with, successful compilation says nothing about whether or not comments are up to date, or if the generated executable is still compatible with existing interfaces.

The problem of describing these interfaces can be at least partially addressed by making use of a strong type system. If the interface is, at the source code level, identified as nothing more than a series of bytes, then there is no syntactic distinction between different input and output strings of equal lengths. The use of type information, however, imposes a much stricter parsing of the underlying byte stream. For example, a three-dimensional vector may represent position or velocity. In a generically-typed setting, with no contextual information, both would appear as indistinguishable triplets of, for example, single precision numbers. If, however, higher level types are created for the concepts of velocity and position, then instances of these concepts are clearly distinguishable from each other. Application of this technique means that code will not compile if it passes three position instances to a function that expects two velocities and a position. While the underlying data, in this case 12 single precision numbers, is identical, the interpretation of those bytes is completely different with the parsing disambiguation offered by type specification. The gain with such an approach is that improper invocation syntax can be caught by the compiler so that the developer can address the problem immediately. Similarly, an invocation target (a data consumer) that claims to be compatible with an interface is less likely to compile if the function does not parse the input properly. That is, if a position vector has named component elements x, y, and z, then an attempt to access a w component will cause an error during compilation. In this case, the developer may choose to define a new interface that supports a position vector with a w component, and, in doing so, explicitly change the interface to his or her code.

2.2 Benefits of Interface Definition by Type
The definition of interfaces further benefits team development as it can provide a helpful perspective on the general strategy of object orientation. By focusing on the borders of an object – its interfaces – we can offer some definition of what that object is. This is difficult when an object’s borders are too porous, that is when there are too many publicly accessible interfaces. When the interfaces are collapsed down to a select few, the method of interfacing with an object comes to better describe how that object is designed to be used. Furthermore, encouraging a minimal number of public interfaces prompts developers to break apart complex objects into sub-objects when they find that they are implementing multiple functionally separable public interfaces. Developers considering the algorithm they wish to implement can break it down into discrete pieces that can be manipulated by considering only their inputs and outputs. That is, one can use type-predicated syntactical rules to reason about fitting multiple objects together into a functional whole. While the interface type definition has no semantic information about what the code does when mapping an input to an output, the syntactical structure the type system imposes on multi-object configurations limits the number of invalid configurations a user might specify. This is a helpful level of abstraction for single-developer design, but it
becomes critical when trying to integrate the work of multiple developers. First, a project can be broken down into multiple parts related by data dependencies, the development of which can be largely accomplished by programmers working independently. More importantly, however, a multi-object configuration syntax makes the job of repurposing code much easier. If a process is broken up into component parts, then each of those parts can be reused in other processes. This repurposing of code can be done very easily at a sufficiently abstract level wherein one can work with “black boxes” that have well-defined inputs and outputs. Given a high level view of how various parts can fit together, any missing links can be custom coded, while the other parts are reused with no changes.

Of particular interest is the fact that a type-based interface description requires no extra effort on the part of the developer. During the course of designing a functional element, a programmer will specify what kind of data is expected as input, and what type of data is generated as output. The individual components of these inputs and outputs can usually be grouped into meaningful structures, or classes, that make working with the functions less cumbersome. This structure, created by the programmer in the natural course of developing their function, can be exploited to gain some understanding of how the function can be used. In essence, one can extract some descriptive content from the functional elements of a program. The advantage in relying on this description is that it is implicitly up-to-date as relates to how it is used by the function itself, because the description, as such, is derived from the same structure used by the functional code when that code is verified by the compiler. In this way, the syntactic description of the interface is valid if the function compiles. Furthermore, the description is generated with no extra programmer effort. To compliment this feature, strongly typed runtime systems often provide the benefits of compiled code inspection. In this fashion, the interface, as described above, of a function can be extracted from compiled code. Thus, there is no chance of source code libraries falling out of date as the source code is not needed to generate the description. The executable code itself is used to generate the description, thus guaranteeing that the syntactic usage description matches the actual code to be run.

Finally, using a robust type system gives the interface system a natural ability to leverage inheritance and polymorphism. The example of a position vector provides a good case study for this concept. If the new position vector type is to be used as an extension of the existing position vector type, then it represents a good opportunity to exploit class inheritance. That is, the original x, y, and z elements remain, but now there is the additional information encoded in the w element. Methods that operate on the original position type may still be validly applied to the augmented position type. In this case, we wish for the augmented position type to be up-cast to the original position type so that it can be operated on by existing methods. In this fashion, base functionality shared by many derived types can be implemented in one place, at the type-specialization level that is most appropriate. Taking advantage of this means that functional code written with one interface in mind can be used to process more specialized inputs without any change to the source code.

3. IMPLEMENTATION

Further analysis of relevant techniques will benefit from the definition of certain terms and design parameters present in a specific implementation of a distributed sensing and computing framework. To this end, the following sections detail the ROCI (Remote Objects Control Interface) system we have developed and used in a variety of single- and multi-robot deployments.

3.1 Software Foundations
A crucial aspect in the development of the framework design philosophy developed above is the relationship between the new software and that which it is built upon. We chose to develop our high level environment on top of an already full-featured platform. In our case, this platform was Microsoft’s .NET technology, which includes a strong type system in the .NET CLR (Common Language Runtime), an object-oriented language in the form of C#, and many varieties of network functionality in the .NET Class Library. Our design then focused both on what functionality we wished to add and that which we wished to remove. Simply put, we want to impose some structure on our developers that is not inherent to C#, .NET, or any existing platform. This structure is a fundamental part of the ROCI philosophy, and is imposed on the ROCI developer as a form of design control that we believe adds a level of reliability to the resultant system. By imposing a prescribed design on developers, we are better able to isolate potential weaknesses and build in error detection and handling functionality.
3.2 The Components of ROCI

ROCI itself is a high level operating system useful for programming and managing sensor networks. The core control element in the ROCI architecture is the ROCI kernel. A copy of the kernel runs on every entity that is part of the ROCI network (robots, remote sensors, etc.). The kernel is responsible for handling program allocation and injection. It allows applications to be specified and executed dynamically by forming communication connections and transferring code libraries to the nodes as needed. The kernel is also responsible for managing the network and maintaining an updated database of other nodes in the ROCI network. In this way, ROCI acts as a distributed peer-to-peer system. Nodes can be dynamically added and removed from the network, and information about these nodes and the code running on them is automatically propagated throughout the system in an on-demand fashion without the need for a central repository.

The control functionality needed by such a kernel is made possible by self-contained, reusable modules. Each module encapsulates a process which acts on data available on its inputs and presents its results on well defined outputs. Thus, complex tasks can be built by connecting inputs and outputs of specific modules. These connections are made through a pin architecture that provides a strongly typed, network transparent communication framework. A good analogy is to view each of these modules as an integrated circuit (IC) that has inputs and outputs and does some processing. Complex circuits can be built by wiring several ICs together, and individual ICs can be reused in different circuits. ROCI modules have been developed for a wide range of tasks such as: interfacing to low level devices like GPS units and cameras, computing position estimates based on GPS, IMU and odometry data, acquiring stereo panoramas, platform motion control, online map building and GPS waypoint navigation.

ROCI modules are further organized into tasks (Fig. 1). A ROCI task is a way of describing an instance of a collection of ROCI modules to be run on a single node, and how they interact at runtime. Tasks represent a family of modules that work together to accomplish some end goal – a chain of building blocks that transforms input data through intermediate forms and into a useful output. A task can be defined in an XML file which specifies the modules that are needed to achieve the goal, any necessary module-specific parameters, and the connectivity between these modules. Tasks can also be defined and changed dynamically by starting new modules and connecting them with the inputs and outputs of other modules.

The wiring that connects ROCI modules is the pin communication architecture. Pin communications in ROCI are designed to be network transparent, yet high performance. Basically, a pin provides the developer with an abstract communications endpoint. These endpoints can either represent a data producer or a data consumer. Pins in the system are nothing more than strongly typed fields of the module class, and so are added to modules with a standard variable declaration statement. Pin communication allows modules to communicate with each other within a task, within a node or over a network seamlessly. The base Pin type will optimize the connection based on whether or not it is local and handle all error detection and handling, bandwidth utilization requirements, and optional buffering. The type system enforces pin compatibility at run time which makes it impossible to connect inputs and outputs of incompatible types.
Importantly, the modules in the system are self-describing so that the kernel can automatically discover their input and output pins along with any user-settable parameters. These features of the ROCI architecture facilitate automatic service discovery since a module running on one ROCI node can query the kernel database to find out about services offered by modules on other nodes and can connect to these services dynamically.

The self-describing behavior of module inputs, outputs, and parameters is achieved automatically through the use of the underlying type system. This is an important element of ROCI’s ability to limit the potential for developer error. In the process of identifying necessary input and output pins, the module developer naturally defines certain data structures that the module takes as input and generates as output. These data structures represent a form of design contract that tells other users what type of input the module can parse, and what type of output it generates. This information is what the pin type system is built upon: a particular type of pin is designed to transfer a particular type of data. These types can then be used to verify potential connections between pins. By relying on type information that the developer necessarily creates by designing module-appropriate data structures, we are able to obviate the need for any separate developer-generated description of a module’s inputs and outputs using mechanisms described earlier.

### 4. DISTRIBUTED EXECUTION

The kind of type-based verification system described above essentially reflects the value of using a carefully considered object inheritance hierarchy in any programming setting. The move to a distributed processing domain, however, brings with it significant obstacles in the way a type system performs. Furthermore, we can modify our original problem domain of distributed computing by identifying certain common usage patterns and priorities particular to sensor networks.

An important usage pattern we have identified is that of the relative infrequency of in-place data modification. While two-way module communications are not uncommon, they are typically used to “ask” a processing module about some data. That is, the processing module returns new data to a calling module. This is in contrast to a model where rough data is processed in-place. Although the “asking” usage pattern may evolve naturally, we encourage it as it presents several benefits over in-place data processing. First, the primary advantage of in-place data processing in a single execution context is the avoidance of unnecessary data copying and memory allocation. By using pointers, the programmer avoids having to copy the input data and allocate a new region of memory to store the result of processing. In a distributed setting, however, there is no possibility of avoiding data copying and memory allocation. The source data must be serialized and copied to the context of the processing routine, which itself must allocate memory to store the result and finally serialize the result back to the caller. Thus, there is no memory usage benefit to be had in designing in-place distributed processing routines. Second, the “ask” model presents the benefits of providing clean opportunities for type differentiation and truly distinct type-specialization levels.

A danger of in-place data processing is that there is not necessarily any way to differentiate raw from processed data. While various ad-hoc methods, such as including a Boolean flag to indicate whether or not a data structure was processed, may suffice in certain situations, we believe that there are significant benefits to making data type the point of differentiability. Primarily, using type to differentiate the states a piece of data may be in provides a standardized method of state identification. That is, the pervasive use of type means that one need never worry about whether or not a data structure has a Boolean variable or an enumeration to indicate its type, or how to evaluate that variable. From a design point of view, type differentiation means that one does not need to verify the state of supplied data. If a function expects a particular type, then it is guaranteed to get that type. No secondary check of a state-indicating field is necessary to verify that the supplied data is valid. This is due to the fact that we can inspect any piece of data to ascertain its type, and stop inappropriate data from going where it is not expected or desired.

The benefit of distinct type-specialization levels is to simplify the application of polymorphism to object-oriented distributed computing as described in the sequel.

#### 4.1 Distributed Type Knowledge and Polymorphism

While polymorphism is extremely useful for a wide variety of applications, most type systems are not built for distributed execution. That is, they entail certain operational requirements in order to maintain certain guarantees of type safety that are not efficient from a distributed computing perspective. A simplified result of this is that when
passing an object to a method that expects an object of a less-specialized type, both functions must know about both types involved. In this case, the calling method, the one that deals with the more-specialized type, is implicitly aware of the less-specialized type due to its presence in the more-specialized type’s inheritance hierarchy. The invocation target, however, may well have been written before the more-specialized type was ever conceived of. In static linking, all of this type awareness is verified at compile time, and all necessary types are rolled into a single executable. In dynamic linking, only metadata is needed for successful compilation; type implementations are stored in separate object code files, but they are present in the same runtime context during execution. In a distributed setting, however, one may want to avoid the need to have a shared execution context where all types are defined. Further, one would like an individual execution context that deals solely with the less-specialized type to have no need to know about any more-specialized types that it does not explicitly make use of. That is, we would like any necessary up-cast to take place in the calling context, so that the invocation target context can be “blind” to the effective polymorphism taking place. There are a few reasons for this, primarily serialization efficiency and memory conservation.

When serializing objects over potentially limited resource connections (like a network), we would like to avoid transferring unnecessary data. The difficulty here is that in most RPC (Remote Procedure Call) systems, both the function performing the invocation and the function being invoked need to know about the types being dealt. In a single-context environment, this is not an issue, but when dealing with multiple contexts that each have their own assortment of types loaded, one must be sure that one does not present a context with a type it is unaware of. The most obviously suboptimal situation is one in which the more-specialized type knowledge is used by the invocation target only to up-cast a more-specialized type of object into the type expected by the procedure being invoked. What is necessary is to trap the invocation in the calling context, and perform any up-casting required by the method being invoked before passing anything to a context that only knows about a less-specialized type.

The second significant advantage of blind polymorphism is memory conservation. In order for a process to make sense of an object passed in as an argument, the context that process is executing in must be aware of the type of the argument. This is problematic when one considers service-oriented processing nodes. Potentially, one would have to recompile the processing routine with a reference to the more-specialized type in the linking stage any time one wished to introduce a new type, causing an undesirable disruption of service. Moreover, the node on which this processing is to be done must have definitions of every type passed in to it, potentially dozens of class libraries given a deep class hierarchy, in the form of object code. Lastly, to maintain type safety, strong type systems generally do not allow the unloading of types in order to maintain type safety. This means that once a type is passed in as an argument, the invocation target must keep that type information in memory for the duration of execution. To “flush” its memory, an execution context must be completely restarted, again introducing a disruption of service.

The need for two-sided type knowledge is to support instance modification and method overriding. If, however, the method being invoked – the data consumer – is not modifying the input data in place, and is not invoking overridden class methods, but is instead consuming the data in a one way fashion, then serializing the more-specialized object is wasteful. Usually a straightforward up-cast is sufficient to strip away an object’s specialization. In other cases, overridden type casts can be used to ensure that the up-cast transforms object-scope data in any way necessary to ensure that it is still valid when accessed through the interfaces of the ancestor type. Similarly, a more-specialized type can be given a method to integrate data from an instance of an ancestor type. In this fashion, methods that modify the more-primitive type can still be used to modify specialized types without both contexts having to be aware of the more-specialized type. While this solution strips away some of the benefits of object oriented design, we feel that it compromises the integrity of object inheritance in a way that is both sensible and advantageous in the distributed computing domain.

4.2 Building the Application

The process of building a standard, non-distributed program follows a series of discrete steps that can be broken up into the main categories of coding, compiling, and linking. Analysis of the distributed computing domain, and ROCI in particular, allows us to specify analogous procedures for building a distributed application in our framework.

First, the process of writing code can be abstracted to a higher level by working at the level of functional blocks with well-defined inputs and outputs, such as ROCI modules. A distributed program may begin by declaring variables of the type of such functional modules. Each module is itself a self-contained, compiled program that can be
reasoned with in a structured fashion due to the model of black boxes with strongly typed I/O. These declared variables must then be instantiated, perhaps supplying construction parameters that are specific to the module type. Next, links between modules—connecting inputs to outputs—can be specified with a command that takes as arguments one input and one output. This gives us a structure of declaration, instantiation, and a form of linking by explicit assignment.

Second, the distributed program must be compiled. Rather than a conversion of readable text to executable code, this step can be seen as more analogous to verifying a program to be executed in a VM (Virtual Machine) with a JIT (Just-in-Time) compiler. In this stage, the system verifies the syntax of the program as well as the type compatibility of arguments supplied in the instantiation statements. If the user passes an initialization parameter of an invalid type, it is critical to catch that error as soon as possible, rather than allowing the potentially costly startup of a distributed program to fail midway through. Note that enforcing type compatibility here requires one stage of linking.

The linking stage of a distributed program is in fact a multi-stage, asynchronous operation. The types specified for the module variables must be examined to determine what the types of the instantiation parameters must be. This is done by first loading the object code containing the implementation of the required types. The underlying strong type system allows us to inspect the compiled code and extract instantiation parameters and their types to ensure that the options specified in the distributed program are valid. The next linking stage involves verifying type compatibility between the inputs and outputs that are specified to be connected. These assignment operations are allowed to fail, but are not discarded if they can not be successfully completed. This is important because these operations may reference interfaces in other programs. It is not a silent failure, but it is a quiet failure: a warning is generated, but execution is not halted. In this fashion, a failed link may cause a program to halt its execution in the case where it is waiting on input from a failed link stage, but it will not halt execution solely because the link failed. If the program can proceed without the successful completion of the link in question, it may do so. Moreover, by allowing links to enter a temporarily failed state, rather than causing a critical exception and shutting down the program, we can sidestep the issue of synchronization between nodes. That is, the requirement that all links be successfully made for an application to successfully load would make circular linking impossible. By allowing failures, we make circular linkages possible by keeping the link stage incremental and asynchronous.

4.3 The ROCI Task Programming Model

The steps briefly outlined in the previous section can be more clearly demonstrated by an analysis of the ROCI task system.

The abstraction gained by treating modules as primitive components allows us to bring compiler-level features to bear on ROCI tasks. Specifically, ROCI provides type checking of the input/output connections between modules, and type checking the parameters that govern the behavior of these modules. Individual module authors are able to decorate class-scope variable declaration statements with attributes that specify whether or not a variable is a startup parameter, or even if it is a control parameter that should be modifiable during execution. These attributes are extracted from compiled code, and are used by the ROCI kernel to expose these variables when appropriate.

Variables marked as startup parameters will be displayed in the ROCI UI when a user wishes to start a task. Type checking is performed as the user enters new values for these parameters, thus making it far less likely that a module will start with invalid parameters. Furthermore, the type of the parameter can be used to intelligently populate the parameter-setting UI by dynamically creating UI elements such as drop-down boxes with only valid values as options, as opposed to a text field for every parameter. Variables marked as control parameters (dynamic over the course of execution) can be modified by a similar interface. A running module can be selected, and any variables marked as control parameters will populate a parameter-setting UI similar to the one described for startup parameters. This functionality, built atop the strong type system in .NET, provides a compiler-like layer of type checking at all phases of execution, while simultaneously making the UI used to interact with a ROCI deployment more intuitive for the end user.

These parameters are dealt with as described by virtue of ROCI interacting with the contexts in which the relevant module types have been loaded. Pin connections, however, represent a somewhat more complex challenge. Since pin connections are completely dynamic (they can be made and broken at any time throughout execution), and involve complex types that are not efficiently represented by plain text (or some other globally understood syntax),
enforcing type safety in pin assignments and allowing polymorphic behavior must involve a different kind of linking. First, pin assignments specified in a task may not be possible at the time of task initialization due to the fact that a remote module to be pinned to may not have started yet. Note that this situation is analogous to an undefined external in traditional linking models. In these cases, we can not verify that the proposed pin connection is valid, so we cache it and continue with local execution (local execution may block until the remote connection is formed, or, in other words, the link can be successfully made). At a later time, we can try to create the pin connection again. If the remote module is active, then we can enquire as to the type of the pin involved in the specified connection, and verify if the proposed connection is valid. This is a process of first checking to see if the types are identical, and if not, if the data sink expects an ancestor class of the data source. This is done by inspecting the inheritance hierarchy of the data source to verify that the sink type is present. If it is, then each time data is passed from source to sink, the source will up-cast the data before serializing it. If this negotiation fails, that is if there is no defined view of the source data that is compatible with what the sink expects, the link fails. In this case, the ROCI kernel raises a warning, but attempts to continue execution.

5. TASK PROGRAM FLOW CONTROL

The notion of task as program allows for varied interesting forms of system-level instrumentation and control. First, by sufficiently isolating individual modules such that they can be treated as atomic operations, we are able to treat tasks as programs built on a language that uses the specified modules as statements. Second, by virtue of its role as provider of all inter-module communications, the ROCI kernel is capable of rich monitoring and control of all data transactions. These two points go towards the notion of program flow control.

Program flow control is primarily governed by the sequence of operations specified in the program. In our case, a schedule of modules makes up the procedural part of a task program. As described above, a task is a collection of concurrently running modules. The order in which these modules run is not explicitly defined, but instead is effectively governed by data dependencies between modules. In general, if module alpha uses data from module beta, then module alpha will block until that data is available, thus creating a very loose schedule in which each iteration of module alpha's processing loop is preceded by at least one iteration of module beta. There are no guarantees on the efficiency of this schedule; if only module alpha uses module beta’s output, then it may be wasteful for module beta to run at a higher rate than alpha.

This issue is addressed by having a task schedule. The task schedule merely specifies a linear sequence of module iterations, but can be leveraged to obtain far greater efficiency than a schedule governed solely by dependency blocking. This schedule is specified in the task XML file as a sequence of module names. The names are checked when the task file is loaded to ensure that all statements in the schedule are defined module names. This schedule can be used simply to eliminate wasted iterations of data producers, but it can also be used to obtain non-obvious gains in overall program efficiency. A schedule can include a bias to run a particular module more frequently than another if it would give the task, taken as a whole, greater efficiency. Furthermore, since this schedule is not encoded in compiled code, it is fully dynamic. That is, a user or automated process can adjust a task’s schedule at runtime to meet changing resource availability or execution priorities.

Such behavior is dependent on information. This information is made available by the instrumentation built into task schedules. The mechanisms that govern the execution of a ROCI task are in good position to monitor the iteration frequency of the task schedule in its entirety, and the resources being used by individual modules. This information can be used to raise alarms when a task frequency drops below a specified threshold, to throttle iteration frequency, or to modify the schedule to make better use of available resources. Furthermore, application specific efficacy metrics can be utilized by task monitoring modules to initiate new schedules that improve efficiency.

The distributed nature of ROCI deployments suggests a form of program flow throttling apart from the usual method of CPU resource allocation: network resource allocation. While the scheduling system can be used to monitor and control the rate at which a task schedule iterates, ROCI’s pin system can throttle network communications on a connection-by-connection basis. Individual pin connections can be monitored to examine the type of data being transmitted, the frequency of transmissions, and the bandwidth used. Both the frequency of transmission and the overall bandwidth used are controllable by the ROCI kernel. This allows a controller, human or automated, to give
network precedence to certain connections, potentially allowing greater system effectiveness given limited resources. Note that by throttling network communications, the speed at which a networked task runs can be controlled. Especially in a schedule-free execution environment, wherein a collection of modules have their iteration frequencies mediated by data dependencies, the throttling of individual connection bandwidth can be used to control the iteration frequency of individual modules. Thus there are two distinct methods of controlling performance in an on-demand fashion based on mediating CPU or network resource allocation.

6. FINAL WORDS

The work described above is representative of our efforts to create a solid software foundation that can support multiple teams of developers working in loose cooperation. The design process began with a forward-looking mentality, and an eagerness to adopt modern trends in software design. While this direction brings with it a learning curve, especially among developers with a practical background in traditional programming, we have found that the concepts and structure of the ROCI system are easily grasped by computer scientists, and actually more intuitive to non-programmers than traditional frameworks based directly on C++-style object-orientation and socket programming. By imposing some loose boundaries on how the development of a ROCI program proceeds, we encourage a stable, consistent design structure. However, imposing a structure on programmers is a proposition rife with opportunities for rejection. If the ideals and concepts that form the foundation of the imposed structure are not in tune with the underlying language and technology, then developers are left with an inconsistent, frustrating, and seemingly illogical development process that they are more likely to try and shortcut. These shortcuts are invariably the beginning of the end for any design paradigm. The prescribed system should be so easy to use, and provide so much built-in functionality, that developers should actually want to use it to design even preliminary, test versions of their programs. In other words, the supplied development system should not be a final software port target, but a truly useful tool that developers want to use.

The goals of intuitive development patterns and feature support can be realized by supporting the salient, productivity enhancing features of the underlying technology. For example, if our developers are writing their code in C#, with its built-in strong type system, then if the supposedly higher-level system they are targeting does not support that type system, there is a potentially disastrous disconnect. If such a valuable feature of the foundation technology is abandoned in the higher level system, then a programmer will have to decide which level of technology they wish to target with their applications. Strong type systems have been shown to aid in rapid development and bug minimization; if the high level system does not also offer those advantages, then a developer would be completely justified in restricting some of their development to the base technology. However, when the high level system offers the same advantages as the underlying technology without requiring too much extra work on the part of the developer, then the choice of which level of technology to support is much easier for the actual developer in the trenches.

The most interesting part of the development of this system has been envisioning it as an extension to existing trends in programming technology. Rather than reinventing the wheel at every stage, we experimented to find out where existing programming technologies let us down, and worked to design conceptually sound mechanisms that efficiently implement the expected functionality (e.g. type support, polymorphism, inter-object communication) in the distributed computing problem domain. We believe that the concepts presented here, such as multiple-context type safety and quiet-failure linking, are important concepts that will prove to be useful as the field of distributed computing moves forward.

ACKNOWLEDGEMENTS

Parts of the work presented here were done with the support of the DARPA MARS2020 program.

REFERENCES


