DSML: A Distributed State Management Library

Ronak Malik, Albert Qi

Abstract

Oftentimes, it is not possible to achieve the desired effect of a system without multiple computers. Whether that is due to hardware constraints or the spatial isolation of a particular component in a system, distributed systems are simply a part of life. They are also hard. The unfortunate realization that the system you are currently working on will have to be distributed is never a fun one, but it is a realization many of us have all too often. The motivation for DSML is to make the process of building certain kinds of distributed systems much easier by providing a simple way to share state.

May 07, 2023

I. Introduction

A. Motivation

DSML[1] is a library intended for use in building distributed systems and provides an easy way to share state among many computers in a system. While there exist many other solutions for distributed systems development (e.g. gRPC, Apache Kafka, etc.), this library fills a particular niche that we feel isn't covered by a standard remote procedure call or event streaming solution.[2] For example, imagine the example of three or more computers that all require information from each of the other computers in the system. Implementing this in gRPC would require implementing a service (possibly two) for every variable you want to expose on every computer in the system. While possible, it would just create code clutter and force the programmer into a great tedium.

Systems like this aren't far-fetched. Rockets with many flight computers each controlling a different sub-system greatly resemble what was just described. One flight computer reading sensor data needs to share that data with the computer doing guidance, navigation, and control, and that computer needs to tell another computer how the thrust vector control should be angled or how the fins should be actuated. Since sensors and actuators are spread all over the rocket, the flow of data isn't one way. Similarly, any system that requires a large amount of shared state just doesn't fit into the model traditional solutions provide.

B. DSML Overview and Goals

DSML provides the ability to easily share state across all of the computers in a system, while keeping network, memory, and computational overhead relatively low in a way that "Just Works"™. Obviously, the programmer is still developing a distributed system and should be aware of all of the problems that

[1]We're obviously great at naming.

[2]It is of course possible to achieve similar results using those libraries, but with greater difficulty.

come with it. DSML does not ignore these problems, and provides the user with some tools to deal with them. It is, however, ultimately up to the user to handle issues of partial failure either within the system or on the network itself. The tools DSML provides to deal with this are discussed in the Features section.

The following are the goals that DSML was trying to achieve. How each of these goals were implemented and the decisions made because of them are discussed in a later section.

1) (Almost) Seamless Distribution of State: The primary goal of DSML was to make sharing state in a distributed system almost seamless. That is, the user should be able to set a variable on one machine and the value of that variable should be available on another machine analogous to passing information between threads. However, it cannot be stressed enough that the user should still understand that they are developing a distributed systems and there are additional challenges associated with that.

2) Versatility with Type Safety: We also thought it important to support many different types so the user can easily use the library without too much extra interpretation work on the user's side. This also gives us the ability to provide loose type safety. DSML provides support for signed and unsigned integers ranging from 8-bit to 64-bit, single and double precision floating points, strings, and variable-sized arrays of any type. Any type more complex than the ones describes can be represented as an array of bytes. The implementation details and limitations of the typing and type safety system are discussed in the implementation and design decisions sections respectively.

3) Pay For What You Use: A defining principle in DSML is a ”Pay for what you use” mindset. If your system isn't distributed, DSML adds no overhead if used as just a centralized state store. If a component is only interested in a single variable, it will only receive updates for that variable. Keeping overhead low and making the library easy to use in all cases was an important consideration when developing the system.

II. Design Decisions

This section will go over the various design decisions and the associated benefits/consequences in DSML. Many are relevant to the user because they affect the way this library should be used.

A. Static Variable Declaration

One of the first design decisions made in DSML was to require that variables were declared statically in a configuration file. This provides many benefits. First, it removes some pernicious ordering problems relating to declaring a variable in one part of the system and wanting to access that variable in another part of the system. While this could have been resolved by adding signaling for variable declaration, this seemed excessively complicated in the face of the more simple and practical solution of just statically defining the variables.

This also allows for some semblance of ”private/public” visibility of variables. You can choose to expose variables to a section of the system by whether or not you include that variable in a component's variable configuration file. Since existence checking of a variable happens locally, if a component doesn't know of the existence of a variable, it won't be able to access it.

B. No Autocode/Code Generation

The next logical decision we had to make was how to get those variables into the system itself. We decided against code generation, partially because that would increase the scope of this project beyond what we could do in the time frame, but code generation would have also required additional steps for the user which we wanted to avoid. Additionally, there isn't anything particularly language specific to DSML other than the API. By simply writing new bindings for another language, it should be easily portable. However, if we chose to integrate the variable file into the code-base using code generation, we would have to write a new code generation tool for every language we wish to support in the future.

However, there are some consequences of not integrating the variable config file into the code, the main one relating to variable and type checking. Since the config file is parsed at runtime, we can't statically check if a variable has been declared or if the type of a variable is correct when it is used. Instead, we opted for a fail-fast system. If a variable is used that wasn't declared or if a variable is used incorrectly (i.e. setting an int variable to a string value), the erroneous function call with throw a runtime error. While this can technically be caught, this is meant to signal an unrecoverable error.

C. Variable Ownership

One of the biggest problems when dealing with shared state is the problem of consistency. It is fundamentally difficult (perhaps even impossible when considering failure) to maintain a consistent state throughout a distributed system if each component of the system is modifying the state. To avoid consistency problems entirely, DSML introduces the property of variable ”ownership”.[3] The owner of a variable is the single source of truth for the value of that variable and is responsible for notifying interested parties if that variable changes. This ensures that only a single system component is changing a variable and there will be no incoherence in the system's state (i.e, one component authoritatively thinks the value of a variable is different than another component). Of course, if the owner becomes inaccessible to a component, the value might become outdated, but the component will be aware of this and will correct the value if the owner comes back online.

III. Features

This section is an overview of the features DSML provides to the programmer.

[3]This also forces the programmer to think about failure modes in their system such as what will happen if the owner of a variable goes down or is inaccessible.

A. Variables

DSML gives the user the ability to define typed variables of signed and unsigned integer types of 8, 16, 32, and 64 bits, single and double precision floating points, strings, and variable sized arrays of any type. More complex data types must be serialized into a DSML compatible format (most likely a byte array). These variables can be set and get using the DSML API. If a variable is set on one computer, the value of the variable can be retrieved on another computer (or the same computer). Similar to a normal multi-threaded application, the values of variables are not available instantly. We still must deal with network and processing delays. DSML provides tools to deal with this though.

B. Signaling

DSML provides the ability to block, indefinitely or with a time-out, on a variable being updated.[4] This allows the programmer to not consume any resources while waiting for a variable update and ensures that the most up-to-date value of a variable is being used. It is important to note that waiting threads are notified whether or not the semantic value of the variable changes. For example, a waiting thread will be notified of an update if the owner changes an integer variable from five to five. There was no real value change, but an update occurred, so waiting threads are notified.

C. Update Requests

A non-owner may ”request” an update to a variable by sending an updated value to the owner, but the variable will not actually be updated until the owner acknowledges the new value and notifies interested parties. This ensures consistency, but might allow the programmer to reduce the number of variables needed. Update requests can be disabled for a variable in the config file.

Currently, DSML does not provide any distributed locking mechanism, so update requests introduce some danger to the programmer. See the section on Future Work for more information.

D. Failure Management Tools

DSML tries not to define policy for failure modes, or define what a failure mode is at all. In some systems, a delayed value might mean failure while in others failure only occurs if a system component becomes entirely unreachable. Due to these possibly differing definitions of failure, DSML simply gives the programmer the tools and information they need to deal with whatever they define as failure. The programmer must critically define what it means for parts of their system to fail and what to do in these cases.

When possible, DSML tries to fail as fast as possible and notify the user of the failure. For example, when expressing interest in a variable, if the connection to a variable owner has failed, the user is notified with a runtime exception. The variable/type checking system follows a similar rule. However, in the cases of ambiguous failure, DSML provides a few different tools to deal with failure management. The primary

[4]It is ill-advised to block indefinitely on a variable being updated because the owner of that variable may have gone down.

tools DSML provides are measures of when each variable was last updated and when the state received the last message from a particular variable owner. This allows the programmer to check if the variable they are using is overly outdated (overly being defined by the programmer) and take action. DSML allows the user to easily attempt re-connection or re-register a socket with an owner. Additionally, DSML allows the programmer to inspect the underlying status of connections to other system components by sending a PING.[5]

IV. Implementation Details

This section will describe how DSML accomplishes what it does. This section will also use more formal distributed systems language to describe the implementation of the system. The word ”client” will be used to describe a machine (or many machines in the case of ”clients”) that is interested in a variable and the word ”server” will be used to describe the machine that owns a variable. Because ownership of variables can be spread throughout the system, a machine might be a client in one context, while in another context it might be a server. This section will assume a simple example of a single variable. That is, one server and possibly many clients. This example trivially extends to more variables by just replicating the ideas or reversing which machines are the client/server.

A. Network Connection

Before describing DSML's protocol, we need to discuss how machines in the system connect. After instantiating a DSML State, the programmer needs to register variable owners with the state. This can be done in two ways. First, the programmer can tell DSML the IP and port of another running DSML instance. Then, DSML will open a TCP connection to that instance and that connection will be managed by DSML. Alternatively, the programmer can establish a connection beforehand using any desired method and pass a file descriptor to DSML which will be used to communicate. This allows the programmer to manage the connection themselves. This is useful if the programmer needs to setup the connection before hand or wrap the DSML protocol in another protocol (e.g. SSL/TLS).

B. Publish/Subscribe

DSML implements a relatively straightforward publish/subscribe protocol to do variable updates. The first time a programmer does a ”get” for a variable, the client will send a message to the server notifying it of interest for that variable. The server will respond with the current value of that variable and will add the client to a list of subscribers for the variable. Whenever the variable is updated on the server, the server will notify all clients of the change. In order to keep network overhead to a minimum, we decided to only send variable updates on change. The programmer can additionally choose to set a time interval where updates are sent out periodically to subscribers to ensure recency.

On update, the variable name size, variable name, data size, and the new value of the variable are sent to the client. This is the only information we need to send (the data size is excessive in cases other than

[5]These are not ICMP packets, rather DSML defined pings.

arrays) because the client already knows the type information of the variable from the configuration file. Additionally, we don't need an operation byte for variable updates because the sockets the server has associated with clients are different than the sockets the server has for receiving variables from other servers (i.e., the server/client functionality of DSML are isolated).

On the client side, an interest packet consists of a bit indicating whether this is an update request or a variable interest request. If it's an interest request, only the variable name size and variable name are sent. If it is an update request, the data size and requested new value are also sent. On success for either of these packets, the server sends back the value of the variable requested.

DSML only uses two additional threads. One for waiting on interest packets/update requests from clients and another thread waiting for variable updates from servers. These threads are spawned based on the needs of the current instance. For example, if a component of the system owns all of the variables and doesn't receive from any other server, then the variable receiving thread won't run. If a component doesn't own any variables, then a server socket won't be opened and the thread monitoring client connections won't run. Overall, DSML is a relatively lightweight library on network and processor usage.

C. Type Checking

Providing generic functions while also type checking at runtime posed an interesting challenge. We thought it would be silly to have a function for every type we wanted to support, so we looked to template arguments. We have a few templated versions of the get/set functions, then let the compiler infer which one to use based on the arguments. For example, if a vector is passed, the vector template function is called. If it is a primitive type, the standard get/set is called. Strings are handled as a special case because we thought it would be a handy function for the user.

Since you can't explicitly store can't store types in C++, we manually type check our stored type against the native type being used by the user. Unfortunately, this kind of templating results in some not very pretty code and some code re-use. We need to have template functions definitions in the header as well to allow the compiler to generate specializations of the templated functions, so this seems unavoidable, or at least we couldn't think of a better way to do it.

D. Data Copy

Although a small detail, the data returned from get() is a copy of the data DSML stores for itself. While this does introduce an additional copy, this is to ensure that the data the user is accessing won't change unexpectedly. This means that return values from get() are stable and the programmer can choose to use outdated values if they wish.

E. Signaling

Signaling is accomplished relatively simply. When a variable update is received on the client that isn't an interval update, a condition variable is notified and all waiting threads are woken up. Waiting on a variable will also send an interest packet for that variable if one hasn't already been sent to make sure that the client is on the subscriber list for that variable.

V. Future Work

Primarily, the code base could use some cleaning up. We've realized that some of our approaches to problems might have not been the best or most efficient. For example, we had an issue where a blocking thread was holding an important lock. So if another thread wanted to acquire the lock, it would repeatedly send a wakeup byte through a pipe until it was able to acquire the lock. We realize now we could have just had the blocking thread unlock before blocking... live and learn. General improvements to documentation and project structure I think would also be beneficial.

We also think it would be useful to be able to load/store the state. This would allow taking a ”snapshot” of the system which can be analyzed or the user could restart the system from the snapshot. This just seems like a useful property to have, and is a huge benefit of having centralized state (in a single component). This was also a motivating reason for the ”Pay for what you use” goal. We think this system would still be useful as a centralized state store for a single computer, multi-threaded system sinces writes to all state variables are atomic and has built in signaling. This load/store property would be useful in this case as well.

Finally, let's talk about distributed locking. We tried to implement a version of distributed locking using acquisition leases, but couldn't get it working in time for the deadline for this project. We think it would be a cool feature (and still have hopes to implement it in the future), but there is a hiatus on that for now. We still believe that DSML accomplishes a lot and is useful without it.

Relevant Links

The implementation, documentation, and programming guide are here.

Acknowledgment

Thanks to Professor Jim Waldo, the professor for CS 262 at Harvard for which this project was developed. Additionally, thanks to Varun Gandhi, the Teaching Fellow for the course, for the feedback on the project and help throughout the semester.