CPSC 626 Final Paper

Building Foundations for Secure Networking in CaMPL

Alexanna (Xanna) Little
alexanna.little@ucalgary.ca

Feb 3, 2025

Abstract

Categorical Message Passing Language (CaMPL) is a concurrent functional programming language being developed at the University of Calgary. Our implementation of CaMPL previously did not support the distribution of processes over multiple devices. This project aimed to apply principles of network security to make it possible for CaMPL programs to run processes on different devices and connect them over a network. The widely used Transport Layer Security (TLS) protocol supports mutually authenticated session key establishment, so it is used to facilitate the connection of distributed CaMPL processes over a network channel. This paper discusses why this and other design decisions were made in the updated implementation of CaMPL, and identifies possible security shortcomings that should be addressed in future development.

Introduction

Security considerations for most systems and applications are often afterthoughts implemented as patches, yet the security of a system is also often taken for granted. Categorical Message Passing Language (CaMPL) is a concurrent functional programming language that is being developed by the research group led by Dr. Robin Cockett at the University of Calgary. This paper will discuss the development of CaMPL and how security considerations have been incorporated into the initial stages of the implementation of networking functionality to ensure that security expectations are actually met.

The programming semantics of CaMPL are defined using category theory. Category theory is an area of mathematics that focuses on "maps," which are relationships between mathematical structures, for example, a function is a map between sets. The Curry-Howard-Lambek correspondence between functional programming semantics and category theory connects sequential functional programs to the maps in a Cartesian closed category [4]. The input and output types of a functional program are objects in the category, and the functional program is a map between those types. The C-H-L correspondence sparked the question of what the categorical semantics of concurrent functional programs might be, and the development of CaMPL is a response to this question [1].

Using a mathematical foundation for the programming semantics of CaMPL has the benefit of allowing us to mathematically prove that CaMPL programs have certain properties. The most valuable property is 'progress' which is the guarantee that programs will never deadlock or livelock [5]. Progress is guaranteed because the configuration of channels that connect processes will not contain a cycle at any point in the execution of a program. For example, three processes cannot all be connected to each other at the same time, but each process can connect to the other two and then pass one of its connections to the next process. An acyclic configuration is enforced by the types of channels, so the guarantee of progress can be made using a simple type check when a program is compiled.

The concept of a concurrent programming language that uses mathematics to guarantee progress in any program is interesting from a theoretical perspective. In practice, such a language would be most useful if concurrent processes could be set up to run on different devices. For example, use cases where a program provides a service for multiple users are designed with the idea that those users are each connecting to the service using their own device. Additionally, it would be reasonable to expect that when one writes such a program in a language designed for that use case, the language would ensure the processes are properly and securely connected. The previous implementation of CaMPL did not provide networking functionality to distribute processes over multiple devices [3, 7]. Also, creating a language feature that allows processes on multiple devices to connect and operate as one program has the potential to create a security nightmare. However, this feature is also important for CaMPL to become a more useful concurrent programming language. These reasons highlight why the project of creating a way for processes to run on different devices and connect over a network *securely* is so important, so this project aimed to apply principles of network security to develop this functionality in CaMPL.

Background

CaMPL uses message-passing concurrency, so processes can pass an output of their computation as a message along a communication channel to another process to use as input. The categorical semantics of CaMPL is defined using a linear actegory which means a category of messages $\mathbb{M}$ *acts* on a category of processes and communication channels $\mathbb{C}$ in two directions [1]. The two actions provide the semantics for messages to be passed in both directions along a channel, so a process can send a message and receive a response along the same channel.

In the category $\mathbb{C}$, the objects are types of channels, e.g. that an int is sent or that the process on one end is forked so the channel needs to be split into two, and a map between objects is a process that uses channels of those types. This means that for two processes to be connected by a channel, they need to agree on the type of that channel. Users can define channel types with multiple 'handles' so that different interactions can happen on a channel of a single type.

The current implementation of CaMPL is available at `https://campl-ucalgary.github.io/`. In a CaMPL program, a main process `run` spawns and connects all other processes used. All user I/O operations are implemented through 'service channels' which are the channels used by the `run` process. This means that any other I/O operations, such as creating a server or connecting to a server as a client, should also be implemented as services.

The most frequently used service channel type in CaMPL is the `StringTerminal` which creates Alacritty terminals to perform I/O operations with the user [2]. The handles for a `StringTerminal` channel are `StringTerminalPut`, `StringTerminalGet`, and `StringTerminalClose` [7]. When a CaMPL program is started, it creates a 'service manager' server that is a TCP socket listening on `0.0.0.0:3000` by default. Then, an Alacritty terminal connected to an 'mpl-client' process is spawned for each service channel of the type `StringTerminal`. An mpl-client connects to the service manager and sends the id of the service channel for which it was created. The service manager accepts the connection and creates a 'service-client' process to handle the message-passing on that service channel from a process in the rest of the CaMPL program. The service-client process is connected over the socket to the mpl-client which handles the interaction with the Alacritty terminal on the other end. A visualization of the resulting process structure is shown in Fig. 1.

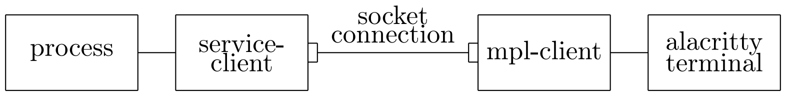

Figure 1: StringTerminal service channels.

Methods

In this project, we have kept the existing user I/O service channels as they were originally and extended service channel functionality to include 'network channels' that connect processes on different devices over a network. This means that any CaMPL programs written without network channels will still function the same as before. We added network channels by creating a new service channel type `ServerClient`, and we modeled them after the `StringTerminal` service channels. The handles for a `ServerClient` channel are `ServerPut`, `ClientPut`, and `ServerClientClose`.

The `ServerClient` service channels create 'proxy processes' similar to mpl-client and service-client. These proxy processes allow the rest of the CaMPL processes to pass messages to them like usual and then handle the interactions with remote CaMPL processes over the socket. A visualization of the `ServerClient` process structure is shown in Fig. 2.

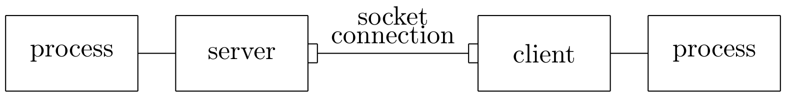

Figure 2: ServerClient service channels.

Network channels were implemented assuming the network is insecure, so Transport Layer Security (TLS) protocol version 1.3 was used to set up mutually authenticated session key establishment to encrypt communication between the proxy processes [9]. TLS requires a server to authenticate itself to a client so that the client can confirm that it has connected to its intended server. However, the relationship between CaMPL processes is more symmetric than most server-client interactions. The only asymmetry arises because a connection must be initialized with one process acting like a server by listening for the incoming connection and the other acting like a client by sending the connection request. After the initial connection is made, a CaMPL 'server' process is equally as incentivized as a CaMPL 'client' process to ensure it only passes messages with the process it is supposed to be interacting with. Another distinction from most common uses of TLS is that CaMPL processes will be set up with the other process's self-signed public key certificate for authentication. This means we needed to use those keys rather than checking against pre-installed certificate authorities.

CaMPL is implemented in concurrent Haskell mostly using the library Control.Concurrent.Async [6]. The TLS implementation we used is originally from the Haskell library Network.TLS [11]. We accessed this implementation through the library Network.Simple.TCP.TLS to set up the TLS connections [8]. We also used the library Data.X509.CertificateStore to handle the certificates used for authentication [10].

Results

Our primary deliverable for this project is an implementation of CaMPL that has network channels that connect processes running on different devices reasonably securely over a network. First, to create the `ServerClient` service channel type, we edited the parts of the CaMPL compiler and assembler that identify service channels and add service instructions to the abstract machine code. Then, we created new service instructions `SHOpenServer` and `SHOpenClient` to create server or client proxy processes when the program begins.

We also needed to edit the assembler so that service instructions to create the proxy processes would actually be added to the abstract machine code when the main `run` process has a `ServerClient` service channel. This part was harder than expected because the compiler does not actually enforce the name of a service channel type. The only thing that matters is the names of the handles which are `ServerPut`, `ClientPut`, and `ServerClientClose`. Additionally, the assembler cannot see what types of service channels `run` actually has because that information is not passed on to the assembler. The only thing the assembler can see is the renamed variable name given to the channel which is the worst thing one could possibly enforce. However, this project is not about the compiler or the assembler and there was not time to change them just for fun, so we are currently checking whether the renamed variable name of the channel starts with `serverclient`. This desperately needs to be fixed as soon as possible.

Finally, we edited the abstract machine to implement the `SHOpenServer` and `SHOpenClient` service instructions. This also required changing the initialization of the abstract machine to read from a `Services/config` file and store environment variables for the host name, port name, keys, and authentication certificates for the TLS connections. This also reads the host name and port name to be used for the service manager. If no `Services/config` file is found, the service manager opens its server on `0.0.0.0:3000` by default and it is assumed that there are no `ServerClient` channels.

In the implementation of `SHOpenServer`, the server proxy process is created by concurrently calling the function `serve`, and we used the function `makeServerParams` to handle the server's credentials and certificate to authenticate the client [8]. To set up the server's credentials, we used `credentialLoadX509` to load the server's private key and public key certificate [8]. To set up the certificate to authenticate the client, we used `readCertificateStore` from Data.X509.CertificateStore [10]. When the proxy process is called, it is raced against the rest of the CaMPL program using `race_` from Control.Concurrent.Async so that it closes when the program ends [6].

In the implementation of `SHOpenClient`, the client proxy process is created by concurrently calling the function `connect` from Network.Simple.TCP.TLS using `concurrently_` from Control.Concurrent.Async[8, 6]. The function `makeClientParams` from Network.Simple.TCP.TLS has bugs that make it unusable to send the client's credentials, especially when `makeServerParams` is used for the server [8]. Instead, we copied and edited the source code for `makeClientParams` and fixed the bugs. Essentially, we changed `makeClientParams` so that it would always send its credential to the server regardless of whether the server sent any information about who the client's certificate should be signed by. This worked well for our use case. We again used `credentialLoadX509` and `readCertificateStore` to set up the client's credentials and certificate to authenticate the server [8, 10].

Evaluation of objectives

The main objective of this project was to make it possible for CaMPL programs to run processes on different devices and connect them over a network reasonably securely. A network channel feature was added to the Haskell implementation of the compiler by creating a new service channel type called `ServerClient`. This feature was added assuming the network is insecure, so TLS 1.3 was used to set up mutually authenticated session key establishment to encrypt communication between networked processes [9]. As mentioned in Results, we found a bug in the `makeClientParams` function from the Network.Simple.TCP.TLS library that we used to set up the TLS connections. The nature of the bug suggested that this library has not been particularly well tested, which is generally not a good sign for security applications. To better fulfill the objective of adding this feature relatively securely, we should consider updating our implementations of `SHOpenServer` and `SHOpenClient` to directly use Network.TLS which is actually a reputable library based on its significantly higher number of forks, contributors, and commits over a similar timeframe as Network.Simple.TCP.TLS. Alternatively, we can further investigate the source code of the functions that we used from Network.Simple.TCP.TLS and test them ourselves to evaluate their integrity.

A secondary objective of this project was to explore how CaMPL might be an interesting language to use to implement security protocols. As mentioned in the Background, the type of a channel specifies the interaction that will take place on it, and processes need to agree on the type of the channel that connects them. This means that the interaction that will take place between two processes is specified by the type of the channel that connects them. This strict typing of channels might be advantageous for the implementation of security protocols as channel types. If a malicious process managed to connect to a legitimate process, it would need to follow the interaction specified by the channel type. If the channel type included a security protocol for mutual authentication before proceeding with the computation, it should prevent a malicious process from successfully impersonating a legitimate process. Due to time constraints of this project, this topic was not explored and remains open for future investigation.

Future work

There are two possible avenues for future work on the networking functionality: one where the implementation of services is kept as it is and another where services are re-implemented entirely. If we re-implement services so that system calls are exposed to the CaMPL programmer, networking functionality could be added as libraries written in CaMPL rather than as something we added to the compiler. This would provide more flexibility to CaMPL programmers for services in general, but it would also be a rather large undertaking.

If we keep services as they are, there are a few security concerns that are worth mentioning. First, a network channel server is started when the CaMPL program starts and is not closed until the entire program is complete. This is because the server proxy process does not have a termination condition other than the fact that it is racing the rest of the CaMPL program. The rest of the CaMPL program only wins the race and cancels the server when it completes or throws an exception. This means that the server is running and listening for connections even after the process that uses it has closed the channel. Although the channel running the server is no longer being used by the program, it would still be best practice to close the server when the channel is closed.

Another potential vulnerability related to services in general is that every CaMPL program creates a service manager server to accept connection requests from mpl-clients even if the program does not use any service channels. Additionally, the service manager accepts all incoming connections and performs no authentication other than matching the service channel id provided by the mpl-client to existing service channel ids. Service channel ids are always generated starting from the same value and in the same way, so they are easily predictable. Furthermore, there is a very small period of time between when the service manager is started by the CaMPL program and when the CaMPL program spawns the mpl-client and Alacritty terminal. This means that a malicious process could likely impersonate an mpl-client as long as it times its connection request to be sent in this small window.

In terms of the functionality of network channels, they are currently implemented so that two processes can connect and send strings back and forth. Ideally, there should also be a way to send arbitrary user-defined datatypes and channel-type handles so that distributed processes can operate more similarly to how local processes would. Adding this functionality will require more significant changes to the compiler, so it remains an area for future work.

Another area we would like to explore is to allow CaMPL servers to accept connections from multiple clients. This can likely be achieved with the current version of the compiler by forking the process with the server channel, but we have not yet investigated whether this functionality works.

Originally, we were also interested in considering use cases where another device does not run a CaMPL process. For example, a program could host a website or webapp. We experimented with this slightly by connecting Alacritty terminals to CaMPL processes across a network, but this still required an mpl-client process to run on the other device to interface between the socket and the Alacritty terminal [2].

Finally, we are interested in investigating how a type check can be performed on CaMPL programs with distributed processes. This is important because we check for progress using a type check. We believe that writing a distributed CaMPL program with all the `run` processes in the same file and including some sort of `super-run` process that connects the other `run` processes will work. We are not sure whether the compiler will allow us to write a `run` process that connects processes along service channels, but it will be easy to test this. If it does not work with the current compiler, we can consider adding this to the type checker. Alternatively, if we rework services, we might be able to do it so that processes can be connected by service channels and use the current type checker.

References

[1] J.R.B. Cockett and Craig Pastro. The logic of message-passing. *Science of Computer Programming*, 74(8):498–533, 2009. doi:10.1016/j.scico.2007.11.005. Special Issue on Mathematics of Program Construction (MPC 2006).

[2] Christian Duerr and Kirill Chibisov. Alacritty. Version 0.15.1. https://alacritty.org/, 2019-2025.

[3] Prashant Kumar. Implementation of message passing language. Master's thesis, University of Calgary, 2018. doi:10.11575/PRISM/5476.

[4] Joachim Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, Cambridge, New York, Port Chester, 1989.

[5] Reginald Lybbert. Progress for the message passing logic. Undergraduate thesis, University of Calgary, 2018. Provided by the author.

[6] Simon Marlow. Control.Concurrent.Async. Version 2.2.5. https://hackage.haskell.org/package/async-2.2.5, 2012-2025.

[7] Jared Pon. Implementation status of CMPL. Undergraduate thesis interim report, University of Calgary, 2021. Provided by the author.

[8] RenzoCarbonara. Network.Simple.TCP.TLS. Version 0.4.2. https://hackage.haskell.org/package/network-simple-tls, 2013-2023.

[9] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor, August 2018.

[10] Kazu Yamamoto and Vincent Hanquez. Data.X509.CertificateStore. Version 1.6.9. https://hackage.haskell.org/package/x509-store, 2013-2022.

[11] Kazu Yamamoto, Vincent Hanquez, and Olivier Cheron. Network.TLS. Version 2.1.9. https://hackage.haskell.org/package/tls, 2010-2025.