Nested Multiparty Session Programming in Go

Author:
Benito Echarren Serrano

Supervisors:
Prof. Nobuko Yoshida
Dr. David Castro-Pérez

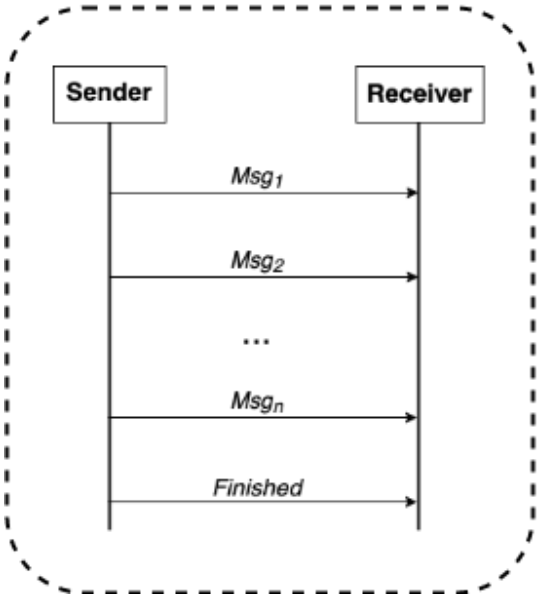
Second Marker:
Dr. Iain Phillips

Contributions

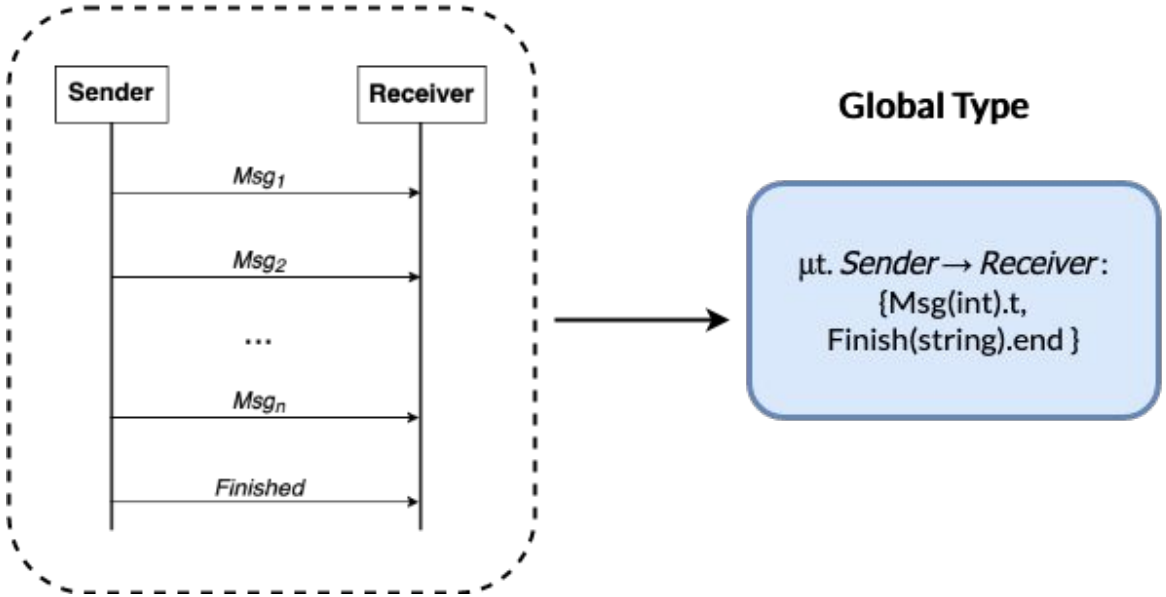
- ➔ Designed and implemented extension to the **Scribble** framework[1]
  - ◆ First practical implementation of **nested session types**
  - ◆ Express **common programming patterns** in **Go**
  - ◆ Express large number of **real-world protocols**
- ➔ Compared **expressiveness** of our extension against previous work [POPL ‘19]
- ➔ Performance evaluation using a **benchmark**

[1] https://github.com/nuscr/nuscr
https://github.com/becharrens/nuscr (fork of repository)

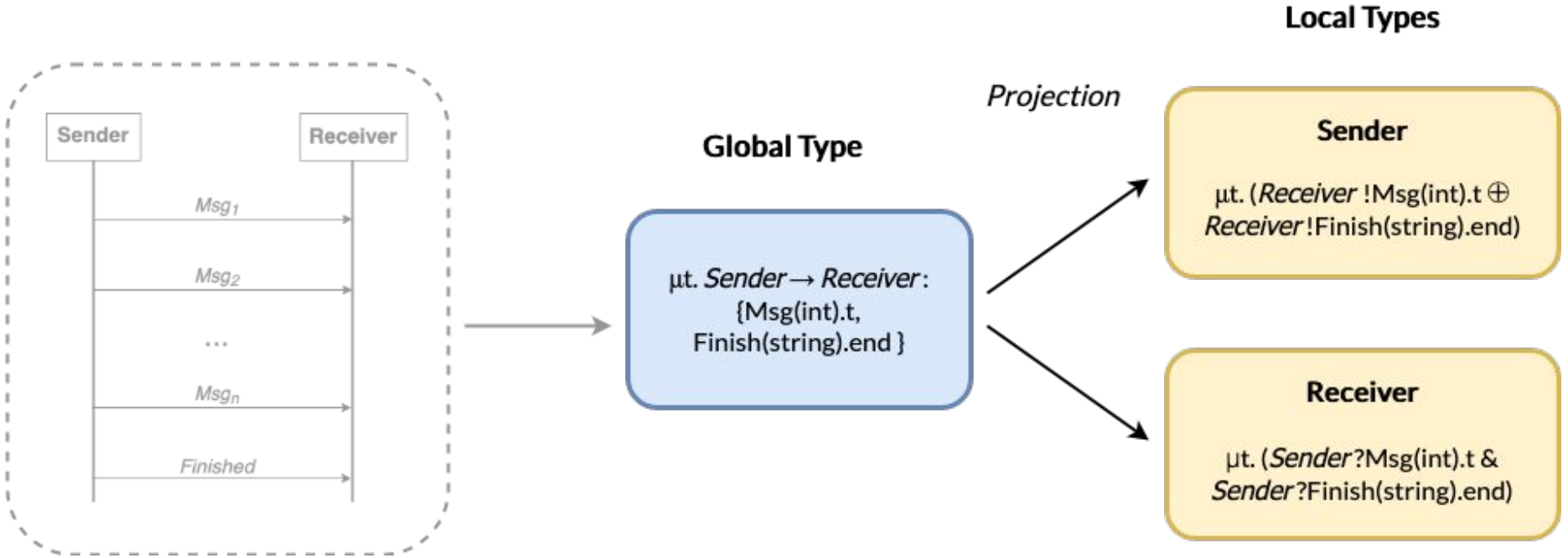
Multiparty Session Types (MPST)



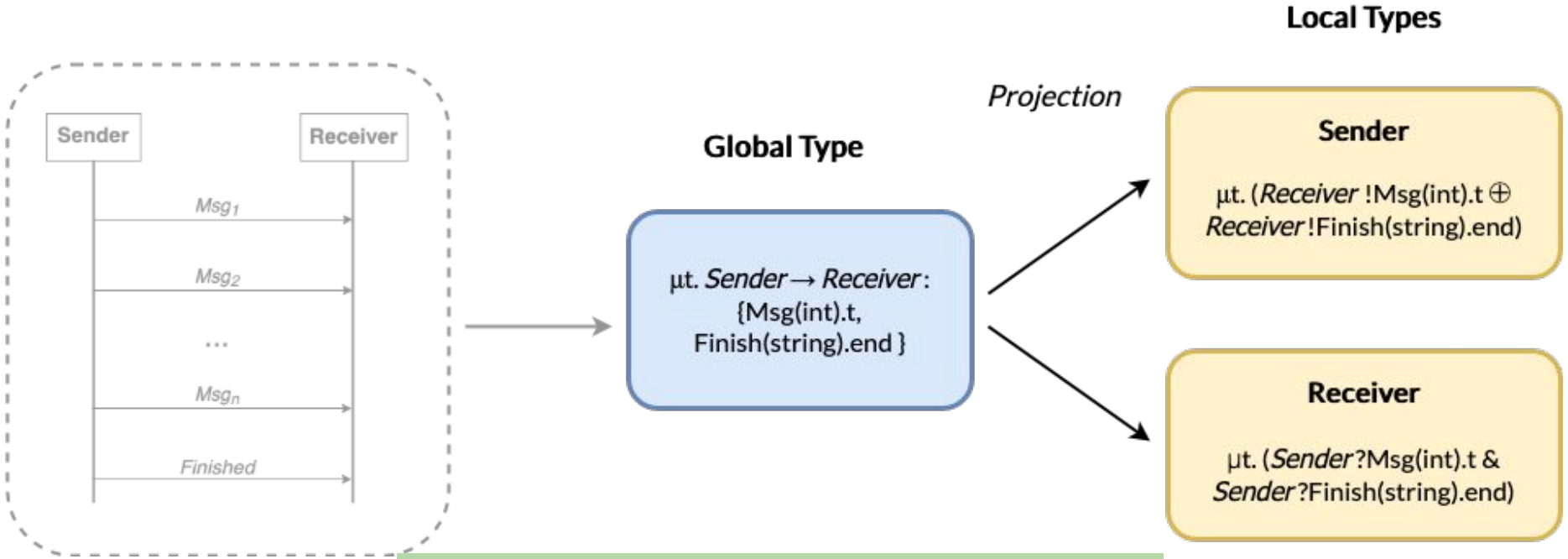
Multiparty Session Types (MPST)



Multiparty Session Types (MPST)



Multiparty Session Types (MPST)

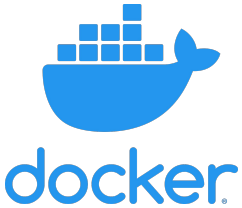


well-typed ⇒ protocol compliance ∧ deadlock freedom

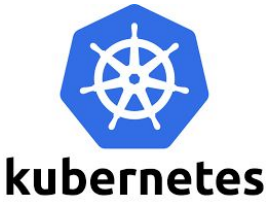
The Go Programming Language

- 13th most popular programming language[1]
- Statically typed, compiled
- Concurrent
  - Goroutines: Lightweight threads
  - Channels: Communication through message passing
- Popular for Cloud Native Applications
  - Scalable, distributed systems

Containers



Orchestration

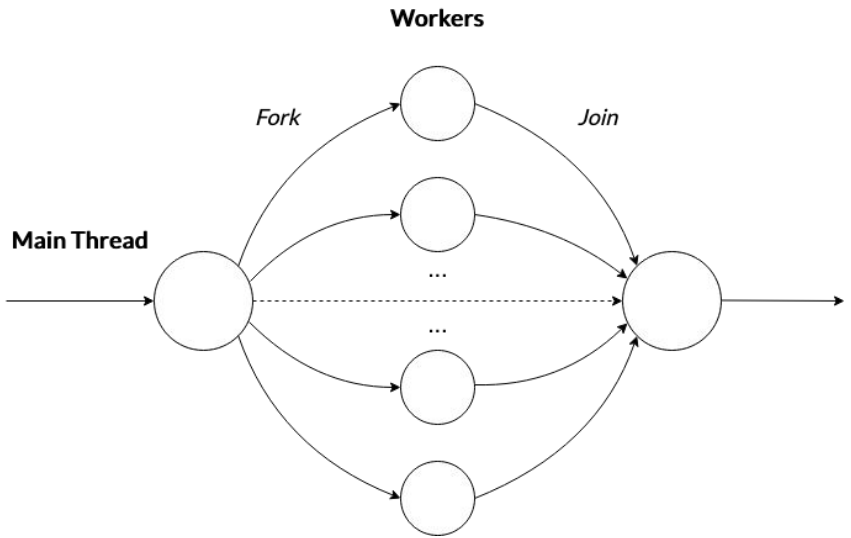


Tracing

[1] http://pypl.github.io/PYPL.html, *June 2020*

Problems

- ➔ In MPSTs, the number of participants **fixed** at the beginning of a session
  - ◆ New participants cannot be introduced
- ➔ This information may **not available** in many practical settings
- ➔ Cannot express common parallel computation patterns
  - ◆ **Fork-join** in **Go**



Solution

- **Nested session types** - MPST theory extension proposed by Romain Demangeon and Kohei Honda [Concur 2012]
  - Allow protocols to call other protocols during their execution
  - Roles in protocol can be **invited** to participate in protocol call
  - Protocol calls can involve **new participants**

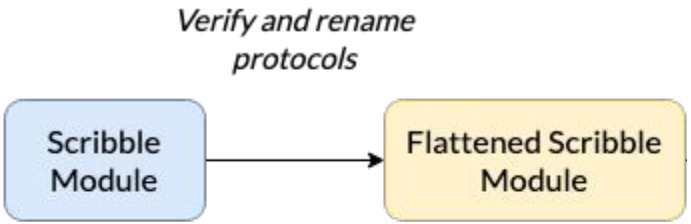
```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        SingleTask() from Master to Worker;
        Result() from Worker to Master;
    }
}
```

Challenges

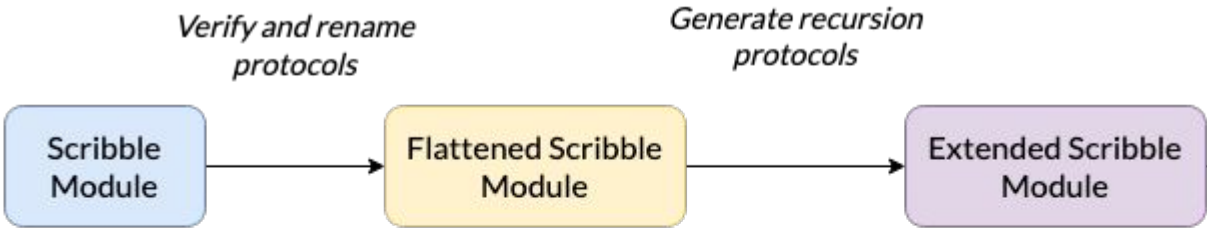
- ➔ Adapting and extending the theory in order to integrate it into Scribble
  - ◆ Formalising definition of **choice** with **invitations**
  - ◆ Formalising definition of **projection** with **full merge**
- ➔ **Returning results** from protocols
- ➔ Supporting **asynchronous communication** with choice and recursion

Nested Scribble Workflow

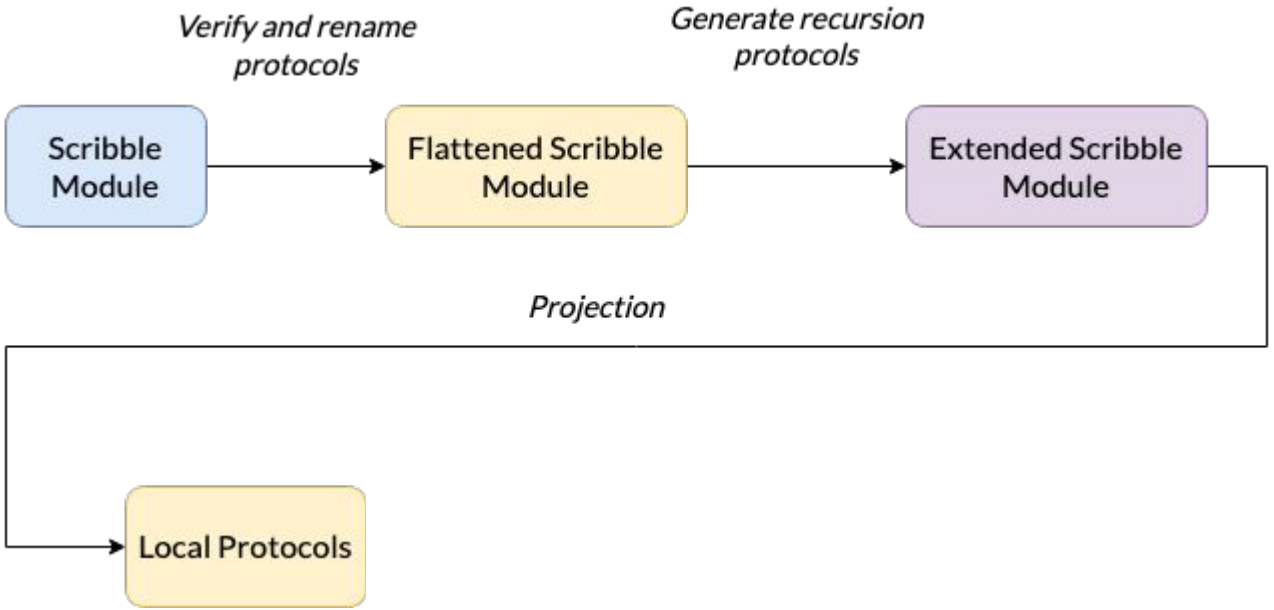
Nested Scribble Workflow



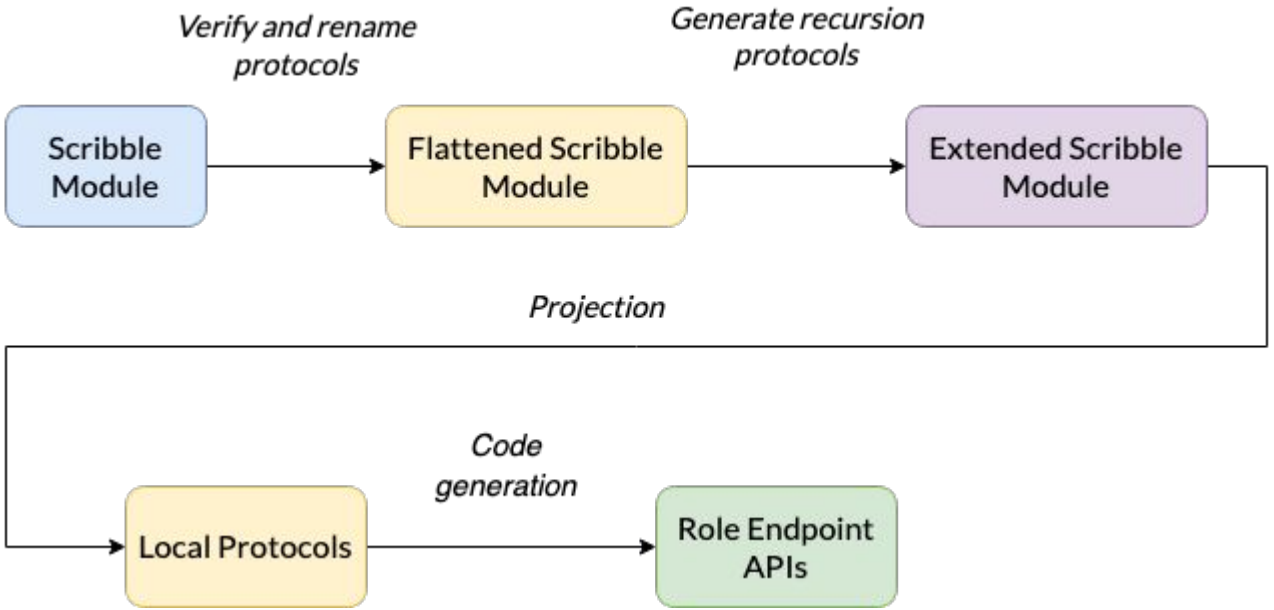
Nested Scribble Workflow



Nested Scribble Workflow



Nested Scribble Workflow

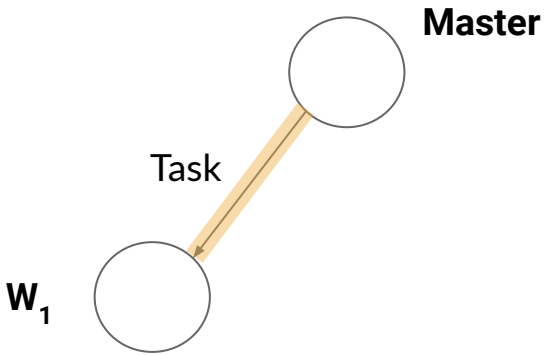


Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```

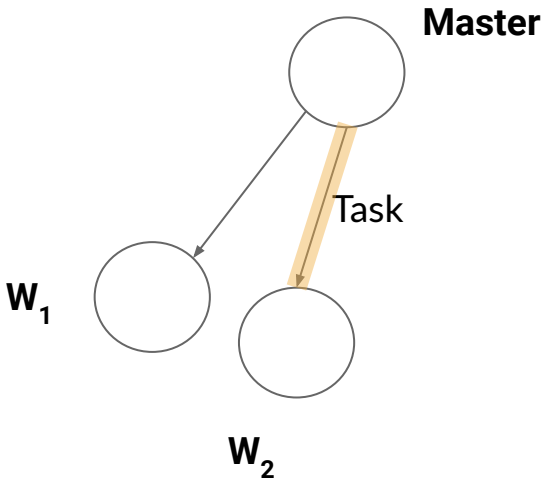
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



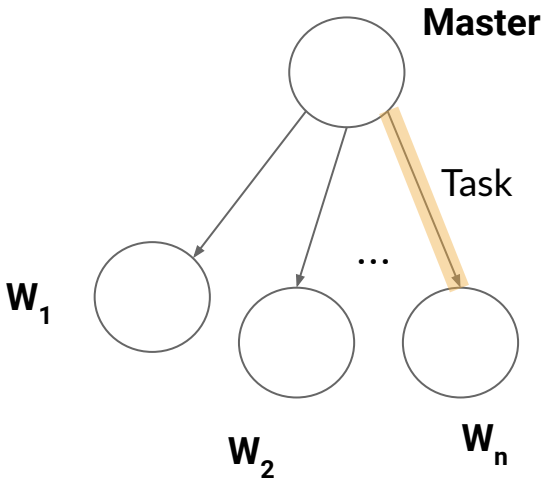
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



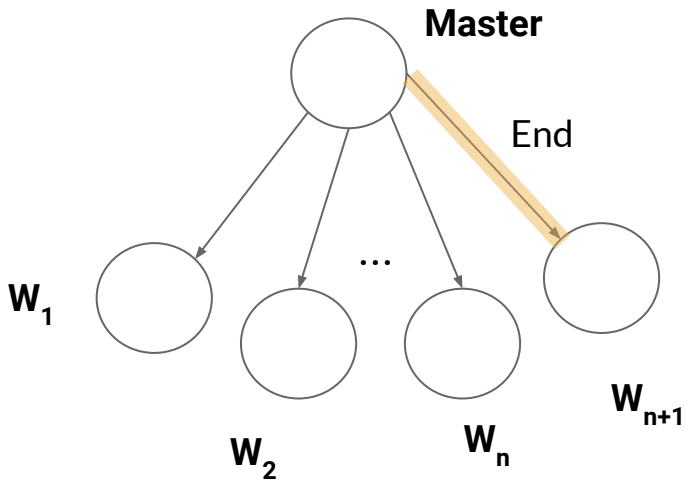
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



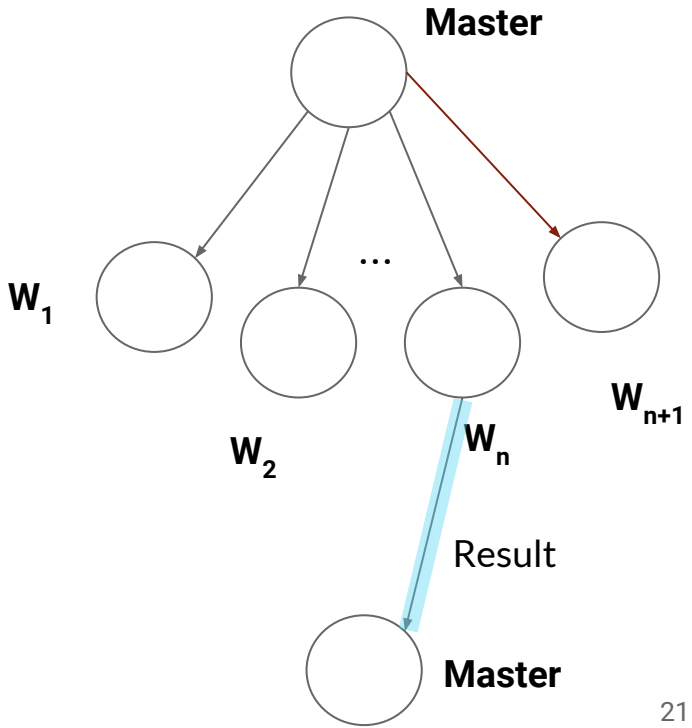
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



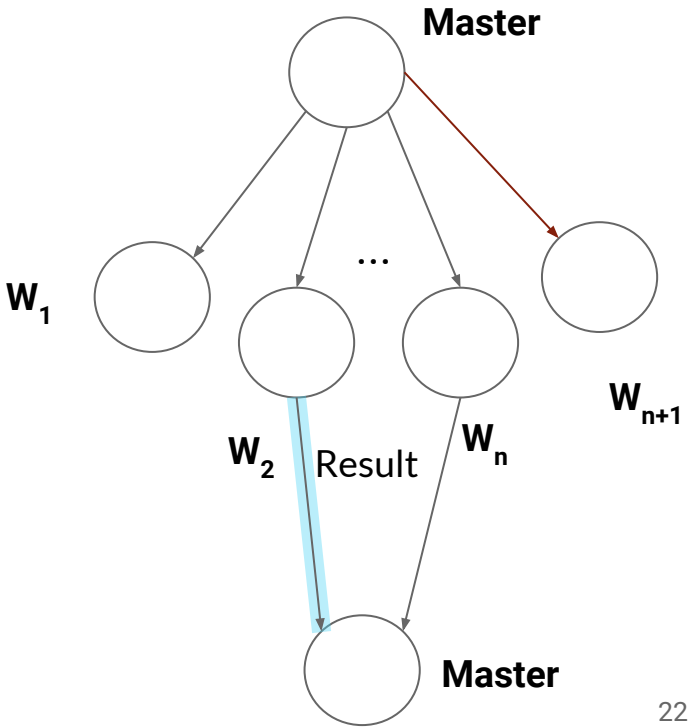
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



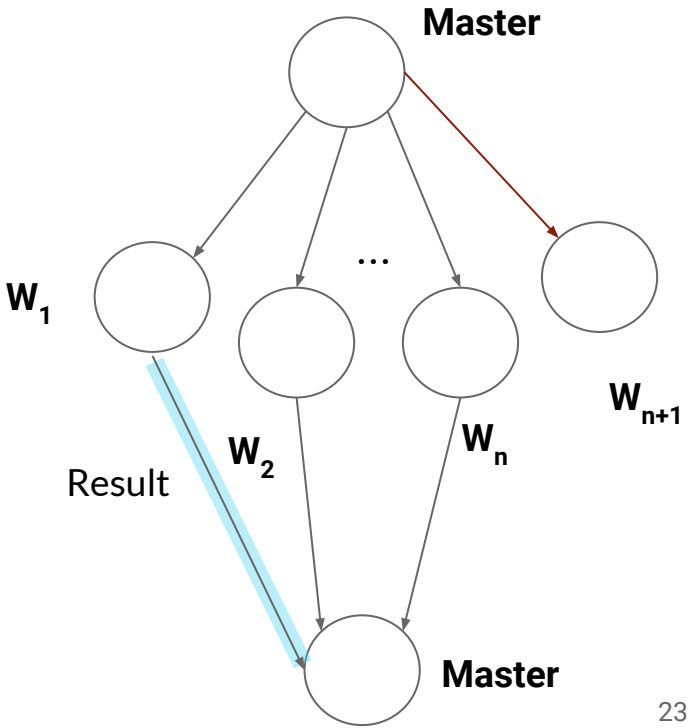
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



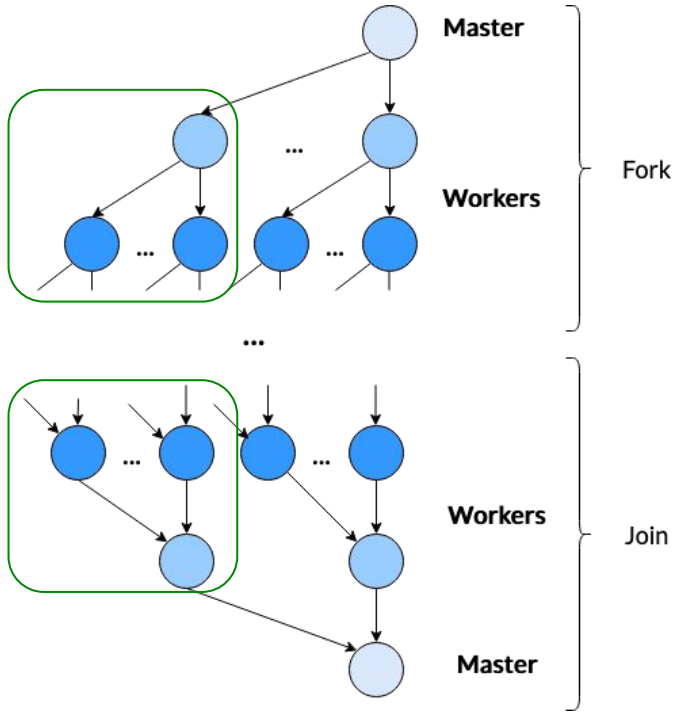
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



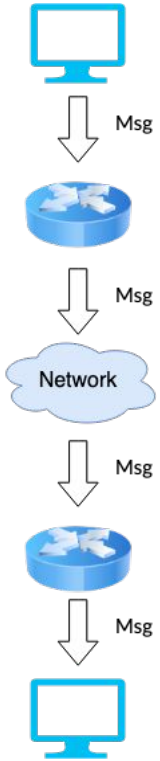
Recursive Fork-Join

```
nested protocol RecFork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls RecFork(M);
        W calls RecFork(W);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol RecForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls RecFork(Master);
        Worker calls RecFork(Worker);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



Routing Protocol

```
nested protocol Forward(role Sender, role Receiver; new role
    Router) {
    Msg(int) from Sender to Router;
    choice at Router {
        Router calls Forward(Router, Receiver);
    } or {
        Msg(int) from Router to Receiver;
    }
}

global protocol Routing(role Start, role End) {
    Start calls Forward(Start, End);
}
```



Routing Protocol

```
nested protocol Forward(role Sender, role Receiver; new role
    Router) {
    Msg(int) from Sender to Router;
    choice at Router {
        Router calls Forward(Router, Receiver);
    } or {
        Msg(int) from Router to Receiver;
    }
}

global protocol Routing(role Start, role End) {
    Start calls Forward(Start, End);
}
```

First interaction in a choice branch need not be a labelled message exchange

Routing Protocol

```
nested protocol Forward(role Sender, role Receiver; new role
    Router) {
    Msg(int) from Sender to Router;
    choice at Router {
        Router calls Forward(Router, Receiver);
    } or {
        Msg(int) from Router to Receiver;
    }
}

global protocol Routing(role St
    Start calls Forward(Start, 
}
```

$$R_1 = R_2 \implies \texttt{IS\_MSG\_FROM}(R_2,\ \texttt{a(S) from } R_1 \texttt{ to } C;\ G)$$

$$R_1 = R_2 \implies \texttt{IS\_MSG\_FROM}(R_2,\ R_1 \texttt{ calls } P(A_1, \ldots, A_n);\ G)$$

$$(\texttt{choice at } B\ \{G_i\}_{i\in I})\downarrow_A^{Env} =$$

$$\begin{cases} \texttt{choice at } B\ \{(G_i\downarrow_A^{Env})\}_{i\in I} & \text{if } A = B \text{ or } A \in \bigcap_{i\in I} \texttt{FIRST\_RECEIVERS}(G_i) \\ \bigsqcup_{i\in I} (G_i\downarrow_A^{Env}) & \text{otherwise} \end{cases}$$

$$\text{if } \forall i \in I.\texttt{IS\_MSG\_FROM}(B, G_i)$$

Routing Protocol

```
local protocol Router@Forward(role Sender, role Receiver; new role
    Router) {
    Msg(int) from Sender;
    choice at Router {
        invite(Router, Receiver) to Forward;
        create(role Router) in Forward;
        accept Sender@Forward(Router, Receiver; new Router) from
            Router;
    } or {
        Msg(int) to Receiver;
    }
}
```

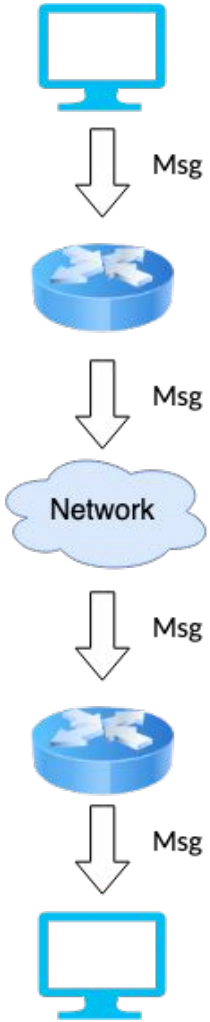
```
nested protocol Receiver@Forward(role Sender, role Receiver; new
    role Router) {
    choice at Router {
        accept Receiver@Router(Router, Receiver; new Router);
    } or {
        Msg(int) from Router to Receiver;
    }
}
```

Invitations are also messages which can be used to communicate a choice made by a role

Code Generation Approach

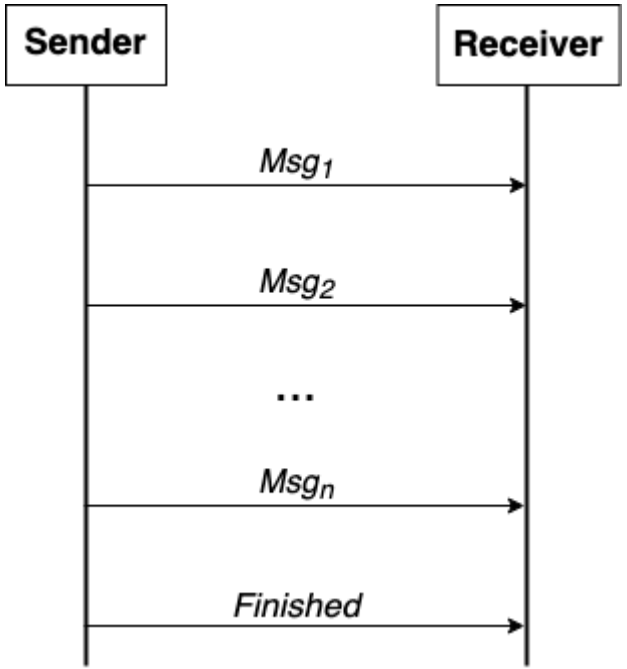
- ➔ Generate role APIs from their **local protocols**
  - ◆ Implementation is **correct by construction**
- ➔ Roles execute as **goroutines** which communicate over **shared memory channels**
- ➔ Protocol implementation defined through **callbacks**
- ➔ Role implementation **returns result**

Routing Protocol Demo



Recursion

- ➔ Difficult to design a correct implementation for protocols combining:
  - Asynchronous communication
  - Choice
  - Recursion



Recursion - Possible Implementation

```
func main() {
    numChan := make(chan int, 100)
    endChan := make(chan string, 1)
    go pipeline.Sender(numChan,
endChan)
    go pipeline.Receiver(numChan,
endChan)
    time.Sleep(1 * time.Second)
}
```

```
func Sender(sendChan chan int,
    endChan chan string) {
    for i := 0; i < 100; i++ {
        sendChan <- i
    }
    endChan <- "Finished"
}
```

```
func Receiver(recvChan chan int,
    endChan chan string) {
    for {
        select {
        case num := <-recvChan:
            fmt.Println(num)
        case endMsg := <-endChan:
            fmt.Println(endMsg)
            return
        }
    }
}
```

Recursion - Possible Implementation

```
func main() {
    numChan := make(chan int, 100)
    endChan := make(chan string, 1)
    go pipeline.Sender(numChan,
endChan)
    go pipeline.Receiver(numChan,
endChan)
    time.Sleep(1 * time.Second)
}
```

```
func Sender(sendChan chan int,
    endChan chan string) {
    for i := 0; i < 100; i++ {
        sendChan <- i
    }
    endChan <- "Finished"
}
```

```
func Receiver(recvChan chan int,
    endChan chan string) {
    for {
        select {
        case num := <-recvChan:
            fmt.Println(num)
        case endMsg := <-endChan:
            fmt.Println(endMsg)
            return
        }
    }
}
```

Generated Output:

```
0
Finished
```

Recursion - Possible Implementation

```
func main() {
    numChan := make(chan int, 100)
    endChan := make(chan string, 1)
    go pipeline.Sender(numChan,
endChan)
    go pipeline.Receiver(numChan,
endChan)
    time.Sleep(1 * time.Second)
}
```

```
func Sender(sendChan chan int,
    endChan chan string) {
    for i := 0; i < 100; i++ {
        sendChan <- i
    }
    endChan <- "Finished"
}
```

```
func Receiver(recvChan chan int,
    endChan chan string) {
    for {
        select {
        case num := <-recvChan:
            fmt.Println(num)
        case endMsg := <-endChan:
            fmt.Println(endMsg)
            return
        }
    }
}
```

Race Condition

Channels are reused throughout all the choices

Extracting Recursion into Protocols

- ➔ **Reusing channels** in different unfoldings of recursion leads to **race conditions**
- ➔ Cannot allocate all necessary channels **statically**
  - ◆ Potentially infinite recursion unfoldings
- ➔ **Allocate channels dynamically** at the beginning of each unfolding of the recursion
  - ◆ **Generate** new **protocols** with the body of each recursion

Recursion Extraction

Before extraction

```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

Recursion Extraction

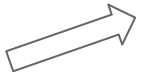
Before extraction

```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

Recursion Extraction

Before extraction

```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```



After extraction

```
nested protocol Pipeline_SEND(role Sender, role Receiver) {
    choice at Sender {
        Msg(int) from Sender to Receiver;
        Sender calls Pipeline_SEND(Sender, Receiver);
    } or {
        Finish(string) from Sender to Receiver;
    }
}
```

Recursion Extraction

Before extraction

```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

After extraction

```
nested protocol Pipeline_SEND(role Sender, role Receiver) {
    choice at Sender {
        Msg(int) from Sender to Receiver;
        Sender calls Pipeline_SEND(Sender, Receiver);
    } or {
        Finish(string) from Sender to Receiver;
    }
}
```

Recursion Extraction

Before extraction

```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

After extraction

```
nested protocol Pipeline_SEND(role Sender, role Receiver) {
    choice at Sender {
        Msg(int) from Sender to Receiver;
        Sender calls Pipeline_SEND(Sender, Receiver);
    } or {
        Finish(string) from Sender to Receiver;
    }
}
```

```
global protocol Pipeline(role Sender, role Receive) {
    Sender calls Pipeline_SEND(Sender, Receive);
}
```

Recursion Extraction

Before extraction

```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

After extraction

```
nested protocol Pipeline_SEND(role Sender, role Receiver) {
    choice at Sender {
        Msg(int) from Sender to Receiver;
        Sender calls Pipeline_SEND(Sender, Receiver);
    } or {
        Finish(string) from Sender to Receiver;
    }
}
```

```
global protocol Pipeline(role Sender, role Receive) {
    Sender calls Pipeline_SEND(Sender, Receive);
}
```

Expressiveness of Nested Protocols

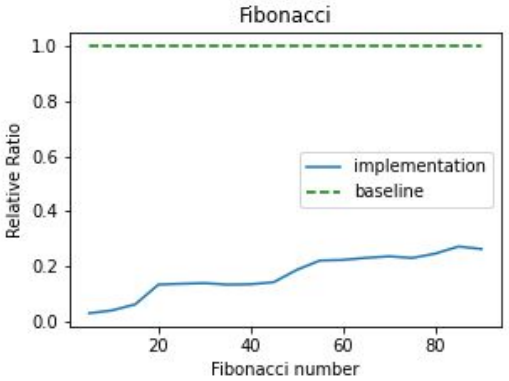
- ➔ In nested protocols, the number of participants within a protocol are **finite** and **cannot change**
  - ◆ New participants introduced through **nested protocol calls**
- ➔ Can only express processes where each step of the computation only involves a fixed number of participants
  - ◆ Can express a protocol to calculate the infinite fibonacci sequence
  - ◆ Cannot express protocols such as the unbounded primesive

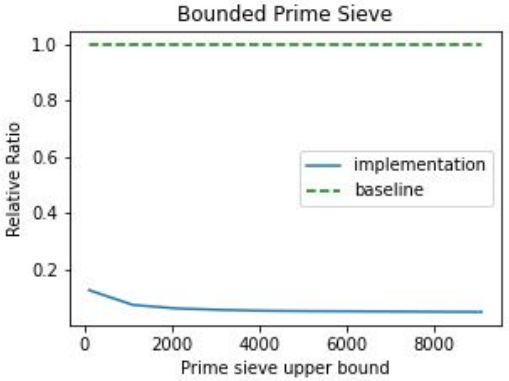
Expressiveness of Nested Protocols

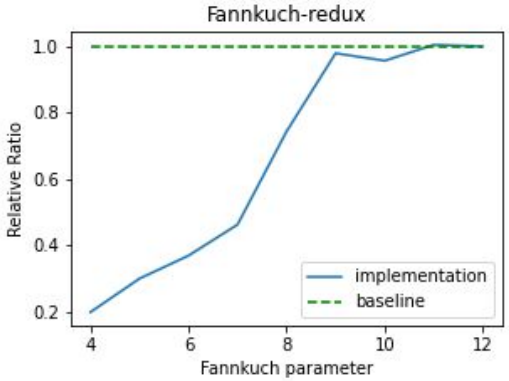
Protocol	Nested Protocols	POPL 2019
`Dynamic Ring`	✓	✗
`Dynamic Pipeline`	✓	✗
`Dynamic Fork-Join`	✓	✗
`Recursive Fork-Join`	✓	✗
`Fibonacci`	✓	✓
`Unbounded Fibonacci sequence`	✓	✗
`Fannkuch-redux`[1]	✓	✓
`Bounded Prime Sieve`	✓	✗
`Unbounded Prime Sieve`	✗	✗

[1]The Computer Language Benchmarks Game

Performance Evaluation



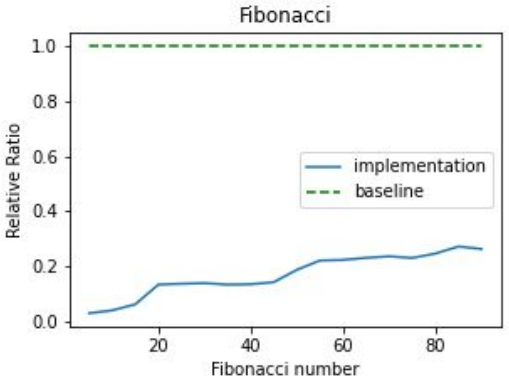


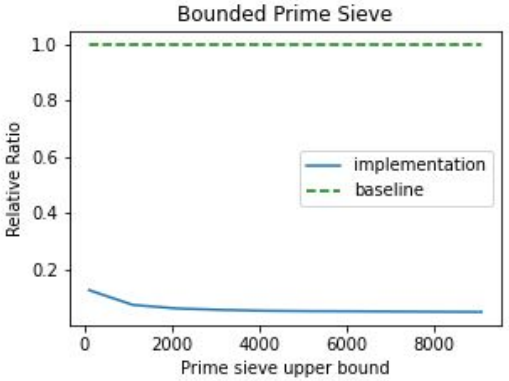


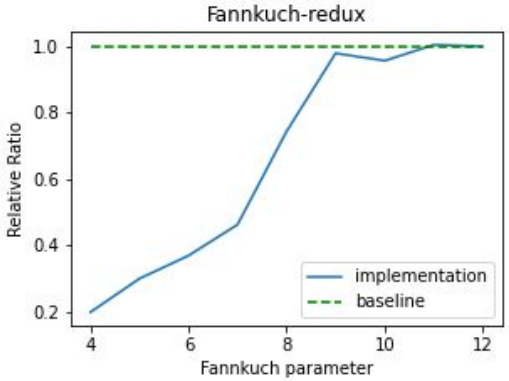
➔ **Benchmark**

- Speedup (t_1/ t_2) of **Scribble** (t_2) vs native Go (t_1)

Performance Evaluation







➔ **Benchmark**

- Speedup (t_1/ t_2) of **Scribble** (t_2) vs native Go (t_1)
- Intel i7- 6700 processor and 16GB RAM

Contributions

- ➔ Designed and implemented extension to the **Scribble** framework[1]
  - ◆ First practical implementation of **nested session types**
  - ◆ Express **common programming patterns** in **Go**
  - ◆ Express large number of **real-world protocols**
- ➔ Compared **expressiveness** of our extension against previous work [POPL ‘19]
- ➔ Performance evaluation using a **benchmark**

[1] https://github.com/nuscr/nuscr
https://github.com/becharrens/nuscr (fork of repository)

Extra Slides

Future work

- ➔ Prove the **correctness** of our implementation
- ➔ **Reduce overheads** of nested protocol calls
- ➔ Implement nested protocols in a **distributed setting**
- ➔ Guaranteeing **termination** in nested protocols
- ➔ Implementing nested protocols using **CFSMs**

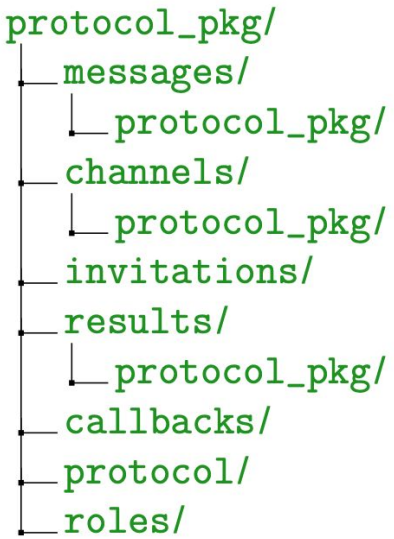
Scope of Protocols

- ➔ Top-level scope
- ➔ Every protocol introduces its own scope
- ➔ Protocols defined within a scope cannot be accessed outside that scope
- ➔ Allow **shadowing** of protocol names
  - ◆ Declaration of a protocol with the same name in a subscope overrides previous definition

Renaming protocols

- ➔ **Flatten** structure of Scribble module
  - ◆ Resolve name clashes between nested protocols in different scopes
  - ◆ Resolve name clashes between global and nested protocols
- ➔ Generate **unique names** for each protocol
- ➔ Update references in protocol calls
- ➔ Simplifies definition of projection
- ➔ Needed for code generation

Implementation Structure



Package messages

- ➔ Generate structs for the different labeled messages exchanged in the protocol
- ➔ Fields in struct correspond to payload of the message

```
type Msg struct {
    Int int
}
```

Package channels

- ➔ Channels used by the roles for labeled message exchanges are stored in a struct
- ➔ Each channel will only be used in one exchange

```
type Router_Chan struct {

    Receiver_Msg chan forward.Msg

    Sender_Msg chan forward.Msg

}
```

Package invitations

- Each role has a struct storing all the channels needed to send and receive invitations
- Invitations consist of:
  - Channel struct
  - Invitation struct

```
type Forward_Router_InviteChan  struct {

      Invite_Receiver_To_Forward_Receiver  chan
forward.Receiver_Chan

      Invite_Receiver_To_Forward_Receiver_InviteCha
n chan Forward_Receiver_InviteChan

      Invite_Router_To_Forward_Sender  chan
forward.Sender_Chan

      Invite_Router_To_Forward_Sender_InviteChan
chan Forward_Sender_InviteChan

}
```

Package callbacks

- ➔ Protocol logic implemented through callbacks
  - ◆ Callback calls interleaved in role implementation
- ➔ Define **interface** with methods that define a role's behaviour, which the user must implement

```
type Forward_Router_Env interface {
     Msg_To_Receiver() forward.Msg
     Done()
     ResultFrom_Forward_Sender(result
forward_2.Sender_Result)
     To_Forward_Sender_Env()
Forward_Sender_Env
     Forward_Setup()
     Router_Choice() Forward_Router_Choice
     Msg_From_Sender(msg forward.Msg)
}
```

Package results

- **Non-dynamic participants** in a protocol will generate a result
  - Mechanism for returning results of computation in the protocol outside of the session
- Generate empty struct - user defines what useful information should be returned

```
type Sender_Result struct {

}
```

Contributions

- ➔ Extended MPST-based framework so it can statically verify the specification of nested protocols
- ➔ Developed **first practical application** of **nested protocols** theory
  - ◆ Increased Scribble's expressiveness with the ability to model many real-world applications
- ➔ Generate correct implementations in Go using its inbuilt concurrency primitives
- ➔ Proposed approach to return results from nested subsessions