Distributed Programming with *Role-Parametric Multiparty Session Types* in Go

Statically-Typed APIs for *Dynamically-Instantiated* Communication Structures

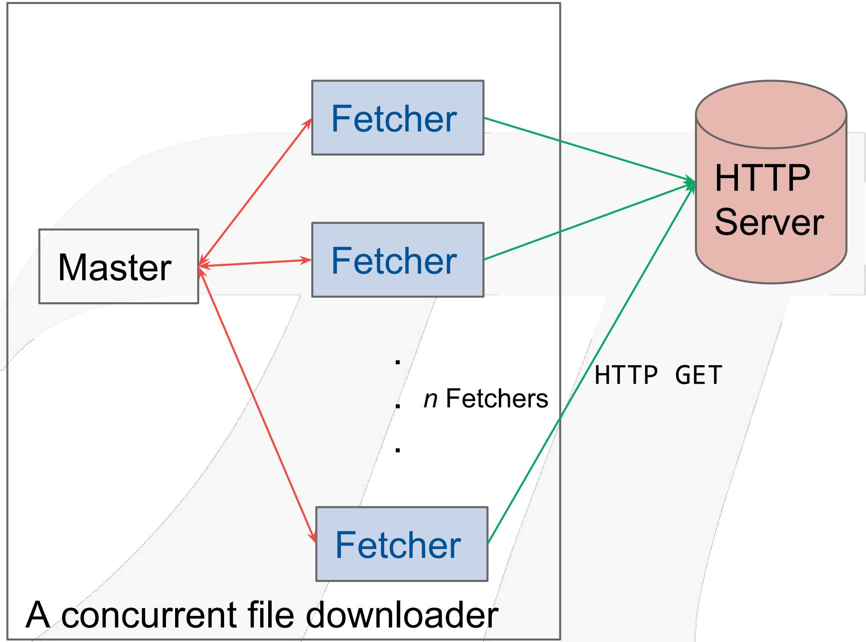
David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, Nobuko Yoshida

A concurrent file downloader in Go

Threads (goroutines)

- Master
- ***n*** of HTTP Fetchers

1. Master send URL/offset to ***n*** Fetchers
2. Fetchers send HTTP requests *x* ***n***
3. Fetchers receive HTTP replies *x* ***n***
4. Master receive data from ***n*** Fetchers



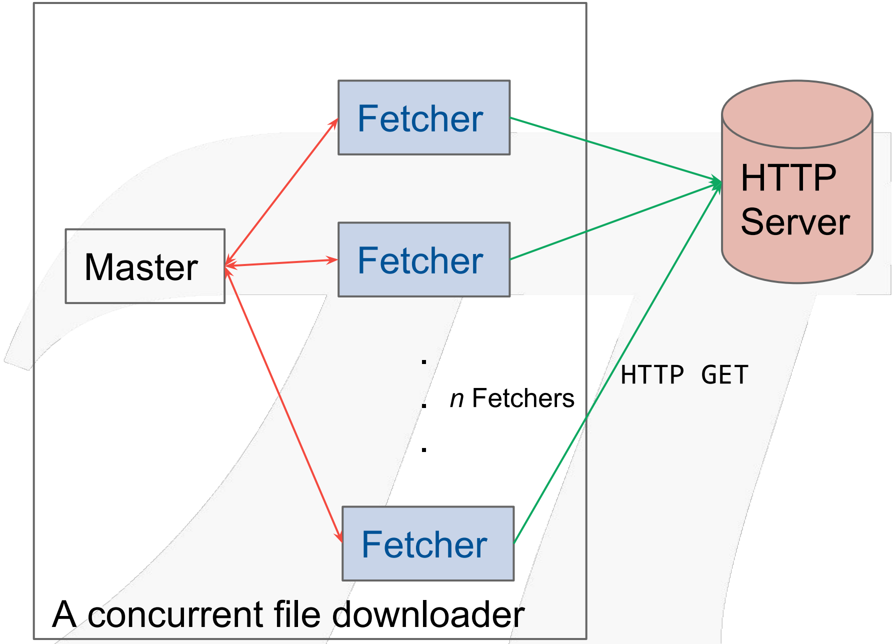
A concurrent file downloader in Go

Threads (goroutines)

- Master
- ***n*** of HTTP Fetchers

In summary

- Message passing over channels
  - Shared memory channels
  - HTTP over TCP channels



A concurrent file downloader

The Go Programming Language

- Statically typed, compiled
- Concurrent
  - *Goroutines*: lightweight threads
  - *Channels*: process calculi inspired communication
- Robust standard library
  - For TCP/HTTP transport etc.
- Popular for *Cloud Native Computing*
  - Scalable, distributed systems (µservices)
  - Concurrency: Inherent asynchrony of distrib. Interactions

Examples:

Containers	**Orchestration**	**Distributed Tracing**
docker	kubernetes	JAEGER

Distributed programming with Go

Go channels: Homogeneously typed (chan T)

- Cannot specify direction of communication
  - e.g. SEND then RECEIVE
- Cannot specify casualty of communication across channels

Distributed TCP/HTTP channels: Generally untyped

Challenges

- Debugging (with concurrency) is difficult [Go user survey 2016]
- Language + library provide not much assistance

How to ensure communication safety & correctness in distrib. sys. in Go?

We offer a solution to the challenges in **Multiparty Session Types**

Multiparty Session Types in a nutshell

Typing discipline for structured communication (POPL'08)

- **Statically** detect communication errors, deadlocks

Global type (or communication protocol)

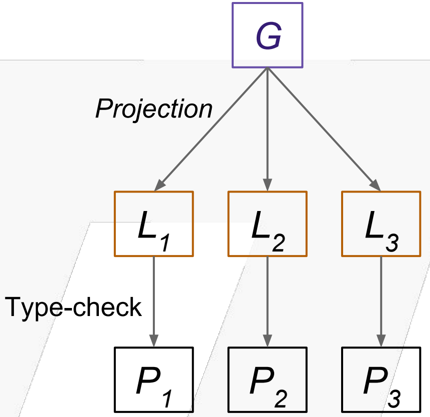
- Describes overall communication structure
- Well-formedness checks

Local types

- Obtained by *projection* onto each role
- Localised view at each endpoint

Processes

- Endpoint implementations
- Type-check against its local types



Traditional top-down *distributed* view of MPST

Concurrent file downloader protocol

Global protocol (for 1 `Fetcher`)

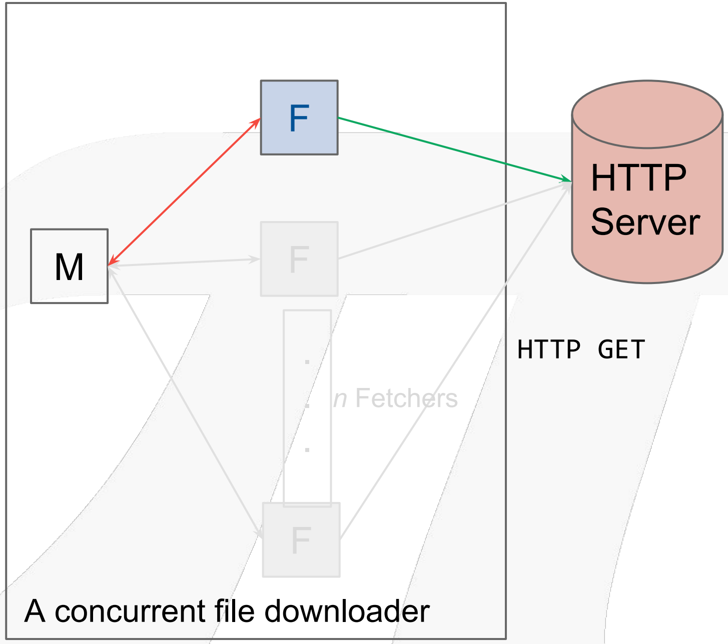
```
Fetch(url) from M to F;
HTTP(req) from F to Server;
HTTP(reply) from Server to F;
Result(data) from F to M;
```

Local protocol @ Master

```
Fetch(url) to F;
Result(data) from F;
```

Local protocol @ Fetcher

```
Fetch(url) from M;
HTTP(req) to Server;
HTTP(reply) from Server;
Result(data) to M;
```



A concurrent file downloader

http://mrg.doc.ic.ac.uk

Concurrent file downloader protocol

Global protocol (for *n* `Fetchers`)

```
Fetch(url) from M to F1; Fetch(url) from M to F2; ...
HTTP(req) from F1 to Server; HTTP(req) from F2 to Server; ...
HTTP(reply) from Server to F1; HTTP(reply) from Server to F2; ...
Result(data) from F1 to M; Result(data) from F2 to M; ...
```

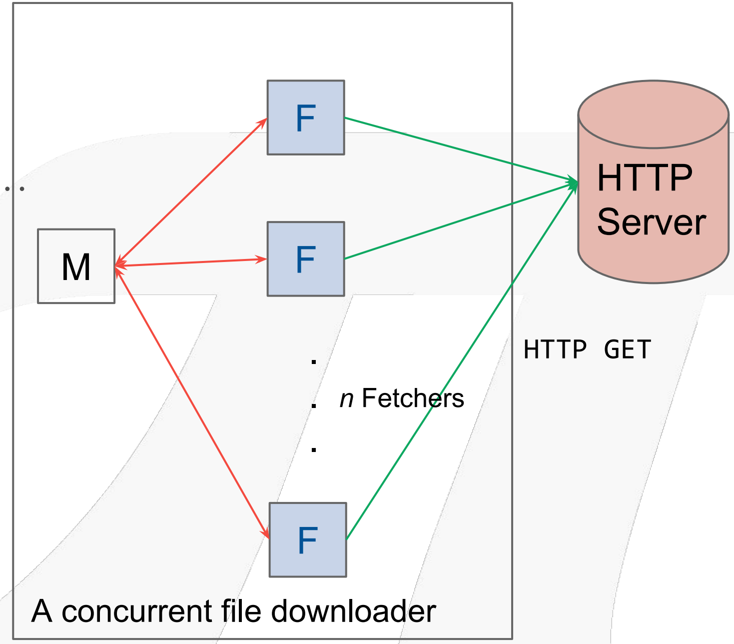
Local protocol @ Master

```
Fetch(url) to F1; Fetch(url) to F2; ...
Result(data) from F1; Result(data) from F2; ...
```

Local protocol @ Fetcher

```
Fetch(url) from M;
HTTP(req) to Server;
HTTP(reply) from Server;
Result(data) to M;
```

Fetcher 1 … Fetcher *n* protocols are the same!



http://mrg.doc.ic.ac.uk

Role-Parametric Multiparty Session Types

When number of participants changes

- Global protocol is different
- Despite core communication structure mostly the same

Intuition: Specify one global protocol and use for ***n*** = 1 or 2 or …

- *Statically* guarantee comm. safety, deadlock freedom as original MPST
- *Dynamically* instantiated communication structure
    - Role parameterised by an index, e.g. `F[1..n] = F[1]...F[n]`

Revised **Concurrent file downloader protocol**

Global protocol (parametric over ***n*** `Fetchers`)

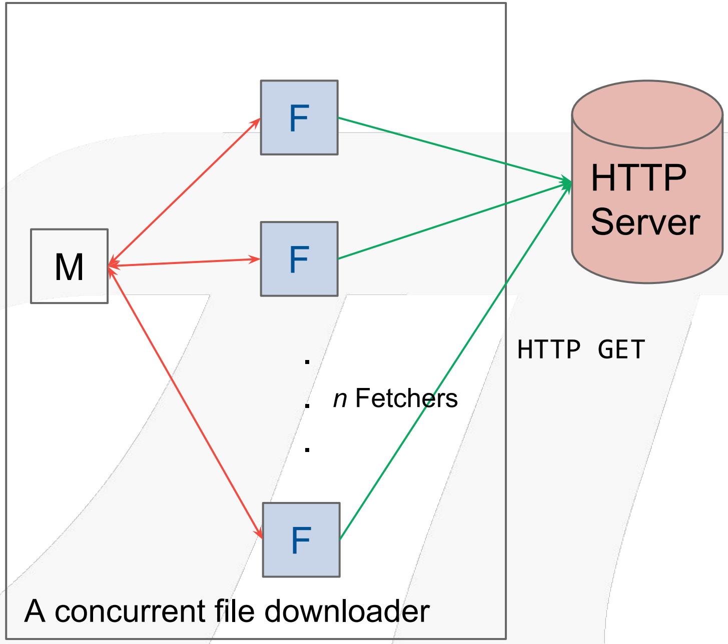
```
foreach F[i:1..n]
      { Fetch(url) from M to F[i];
        HTTP(req) from F[i] to Server;
        HTTP(reply) from Server to F[i];
        Result(data) from F[i] to M; }
```

Local protocol @ Master

```
foreach F[i:1..n]
      { Fetch(url) to F[i]; Result(data) from F[i]; }
```

Local protocol @ Fetcher [1..*n*]

```
Fetch(url) from M; HTTP(req) to Server;
HTTP(reply) from Server; Result(data) to M;
```



Role variant

Role variant are *unique kinds* of endpoints

`{M, F[1..n], Server}`

If `F[1]` sends an extra request

HTTP HEAD to Server to get total size

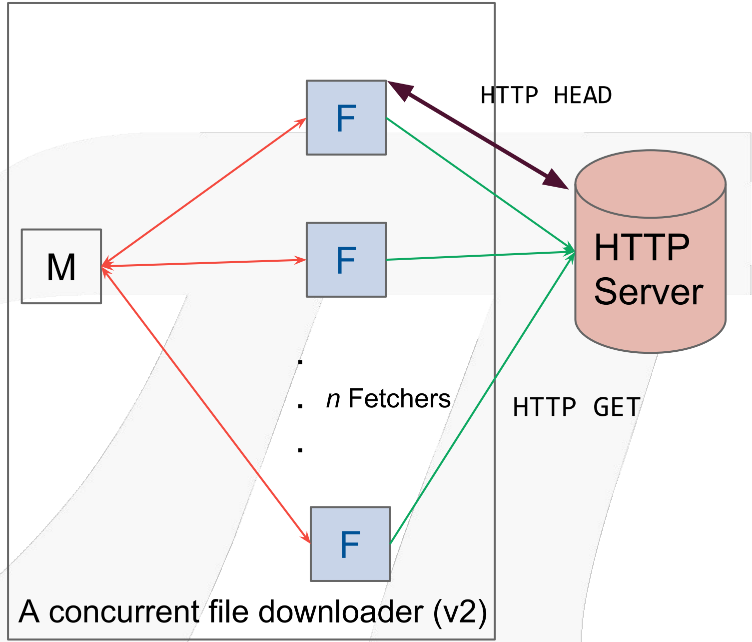
Then acts as a normal `F`

The role variants are:

`{M, F[1], F[2..n], Server}`

→ `F[1]` and `F[2..n]` are different endpoints

Inference of role variants (indices): formulated as SMT constraints for Z3



A concurrent file downloader (v2)

The Scribble-Go framework

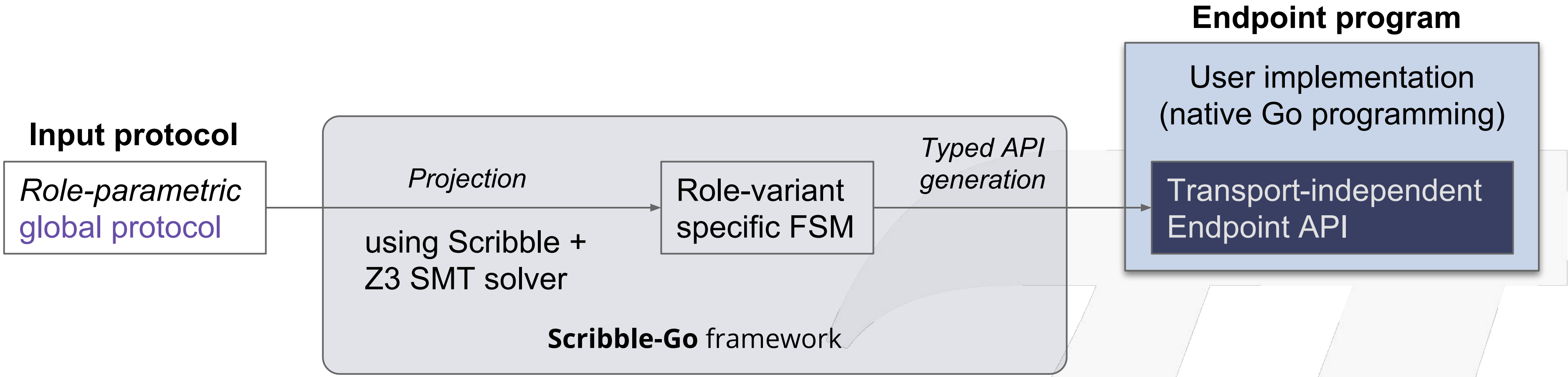
Scribble project (scribble.org)

- Protocol specification language & verification framework
- Practical incarnation of (original) MPST
- Collaboration with industry: RedHat, Cognizant, OOI
- Python [RV'13], Java [FASE'16,'17], Scala [ECOOP'16,'17], Erlang [CC'17], F# [CC'18]

`Scribble-Go`

- New theoretical & implementation extension of Scribble
- Adds role-parametric protocol support
- **Endpoint API code generation** for message passing programming

Scribble-Go workflow



Input protocol

Role-parametric global protocol

Projection

using Scribble +
Z3 SMT solver

Role-variant
specific FSM

Typed API generation

Scribble-Go framework

Endpoint program

User implementation
(native Go programming)

Transport-independent
Endpoint API

1. Write a *role-parametric* global protocol
2. Select endpoint *role variant* to implement (e.g. `Fetcher`)
3. Use **Scribble-Go** to project and generate **Endpoint API**
4. Implement endpoint (e.g. `Fetcher[3]`) using the **Endpoint API**

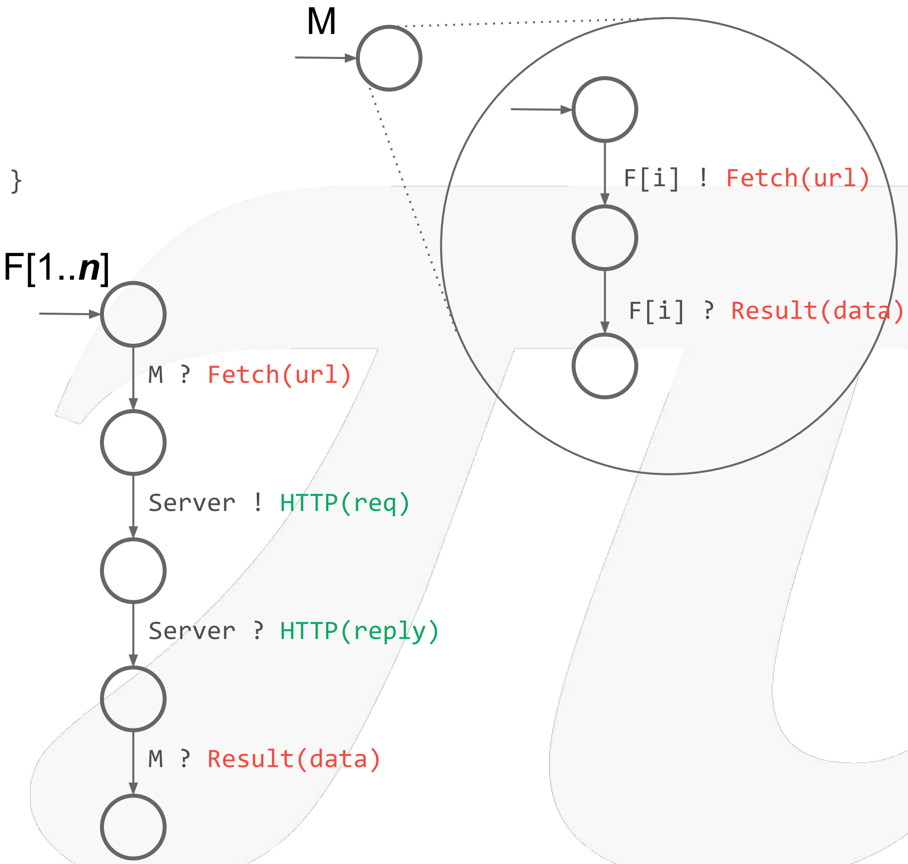
Role-variant local protocol as FSM*

Local protocol @ Master

```
foreach F[i:1..n]
      { Fetch(url) to F[i]; Result(data) from F[i]; }
```

Local protocol @ Fetcher[1..*n*]

```
Fetch(url) from M; HTTP(req) to Server;
HTTP(reply) from Server; Result(data) to M;
```

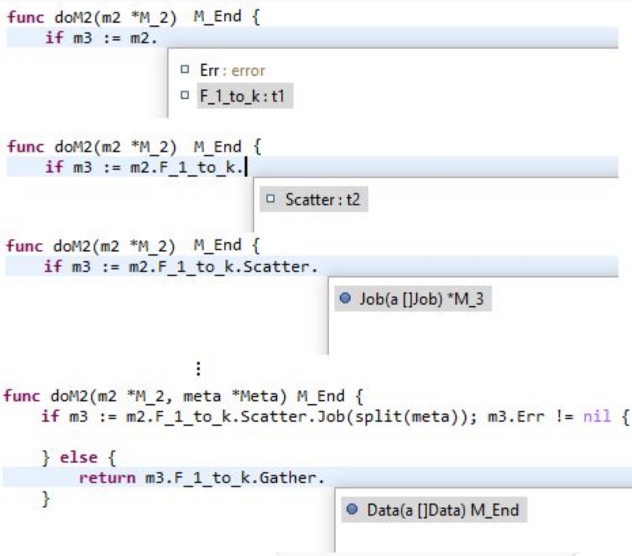


*More accurately, *Communicating* FSM

Endpoint API generation and usage

FSMs from local protocols → Message passing API

- Fluent-style
    - Every state is a unique type (struct)
    - Method calls (communication) returns next state
- Type information can be leveraged by IDEs
    - “dot-driven” content assist & auto complete



```
func doM2(m2 *M_2) M_End {
    if m3 := m2.
```

```
func doM2(m2 *M_2) M_End {
    if m3 := m2.F_1_to_k.
```

```
func doM2(m2 *M_2) M_End {
    if m3 := m2.F_1_to_k.Scatter.
```

⋮

```
func doM2(m2 *M_2, meta *Meta) M_End {
    if m3 := m2.F_1_to_k.Scatter.Job(split(meta)); m3.Err != nil {

    } else {
        return m3.F_1_to_k.Gather.
    }
```

Endpoint API generation

- Generated API is **transport independent**
  - Presented as *generic* message passing communication methods
  - Lightweight runtime abstracts:
    - Shared memory transport (~Go channels)
    - TCP transport (via wrapper of Go's `net` package)

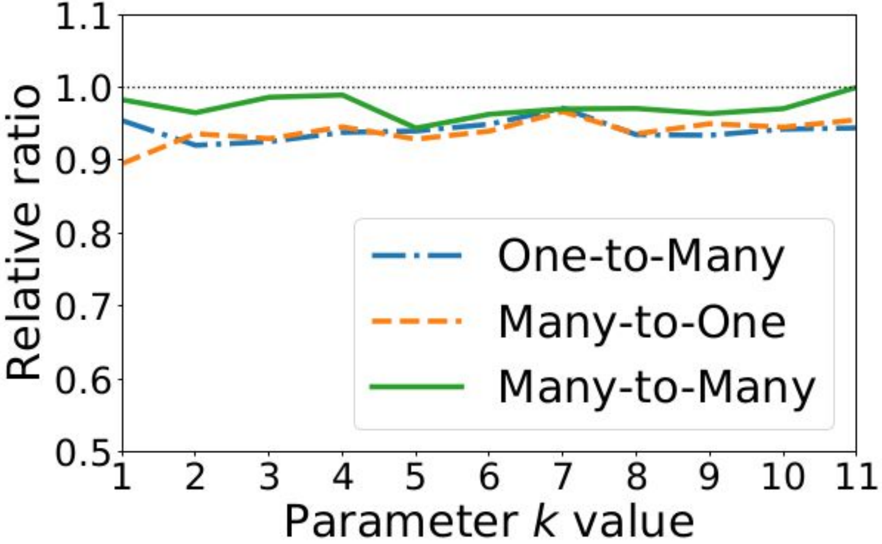
- Also provides **channel passing** communication!
  - Over shared memory transport
  - Transparent to user

→ Send `Protocol@Role` as payload in Scribble

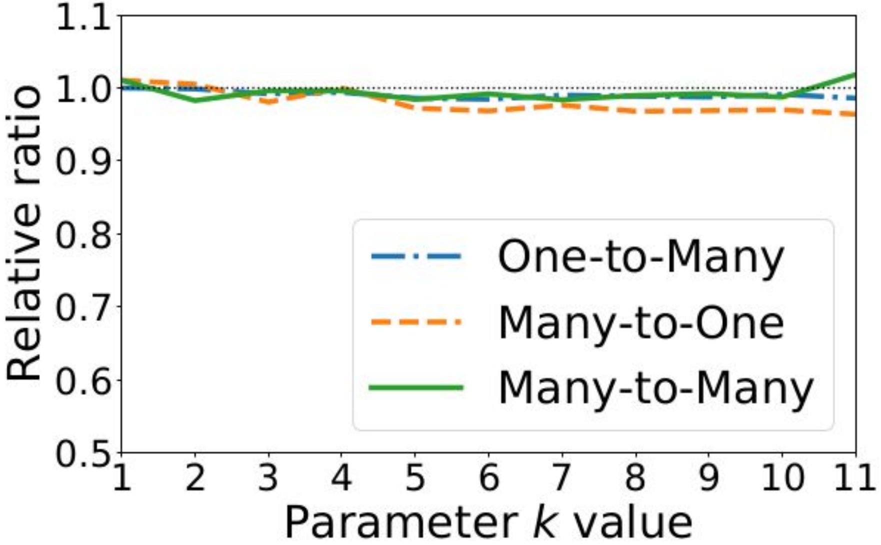
```
Message(Protocol@Role) from Alice to Bob;
```

Evaluation: runtime overhead

Shared memory transport

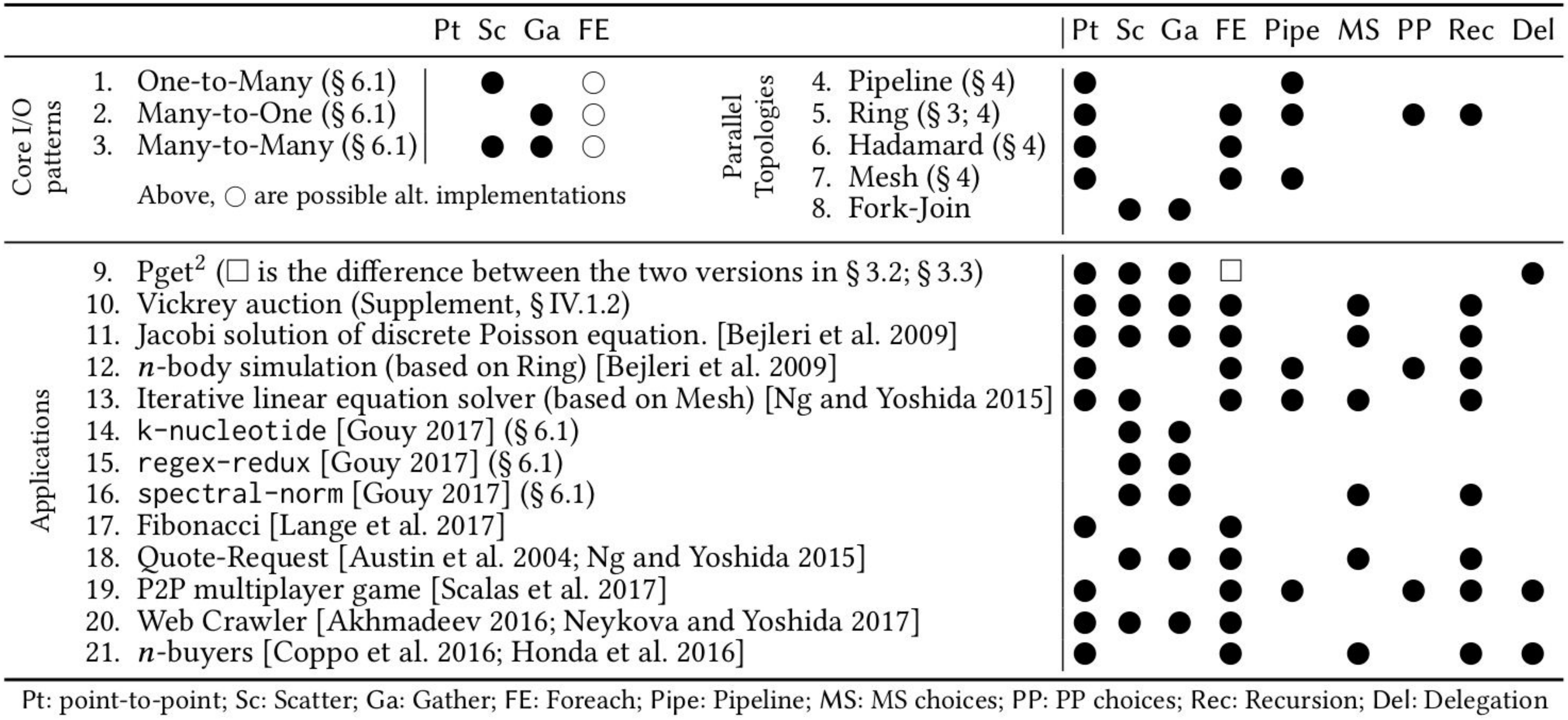


TCP transport



Relative ratio: execution runtime compared to native
1.0 = same as native

Evaluation: expressiveness



		Pt	Sc	Ga	FE
Core I/O patterns	1. One-to-Many (§ 6.1)		●		○
	2. Many-to-One (§ 6.1)			●	○
	3. Many-to-Many (§ 6.1)		●	●	○

Above, ○ are possible alt. implementations

		Pt	Sc	Ga	FE	Pipe	MS	PP	Rec	Del
Parallel Topologies	4. Pipeline (§ 4)	●				●				
	5. Ring (§ 3; 4)	●			●	●		●	●	
	6. Hadamard (§ 4)	●			●					
	7. Mesh (§ 4)	●			●	●				
	8. Fork-Join		●	●						
Applications	9. Pget^2 (□ is the difference between the two versions in § 3.2; § 3.3)	●	●	●	□					●
	10. Vickrey auction (Supplement, § IV.1.2)	●	●	●	●		●		●	
	11. Jacobi solution of discrete Poisson equation. [Bejleri et al. 2009]	●	●	●	●		●		●	
	12. *n*-body simulation (based on Ring) [Bejleri et al. 2009]	●			●	●		●	●	
	13. Iterative linear equation solver (based on Mesh) [Ng and Yoshida 2015]	●	●		●	●	●		●	
	14. `k-nucleotide` [Gouy 2017] (§ 6.1)		●	●						
	15. `regex-redux` [Gouy 2017] (§ 6.1)		●	●						
	16. `spectral-norm` [Gouy 2017] (§ 6.1)		●	●			●		●	
	17. Fibonacci [Lange et al. 2017]	●			●					
	18. Quote-Request [Austin et al. 2004; Ng and Yoshida 2015]		●	●	●		●		●	
	19. P2P multiplayer game [Scalas et al. 2017]	●			●	●		●	●	●
	20. Web Crawler [Akhmadeev 2016; Neykova and Yoshida 2017]	●	●	●	●					
	21. *n*-buyers [Coppo et al. 2016; Honda et al. 2016]	●			●		●		●	●

Pt: point-to-point; Sc: Scatter; Ga: Gather; FE: Foreach; Pipe: Pipeline; MS: MS choices; PP: PP choices; Rec: Recursion; Del: Delegation

Fig. 15. Role-parametric protocols for communication patterns, topologies and applications in Scribble-Go.

Related work

Parameterised MPST [Denielou et al., LMCS'12], Pabble [Ng et al. SOCA, CC'15]

- Single combined local protocol
- Unsuitable for distributed programming
- Or requires special runtime to handle indices (e.g. MPI)

Verification of msg-passing Go programs [Ng et al., CC'16; Lange et al. POPL'17, ICSE'18]

- Bottom-up approach (type inference)
  - No assistance to programmers
- Limited to Go channel communication; no channel passing support

Conclusion

Scribble-Go: A framework for communication-safe distributed programming

- Based on Role-Parameterised Multiparty Session Types
    - Number of roles dynamically instantiated
    - Statically guarantees communication safety, deadlock freedom
- Tool chain
    - Input role-parameterised global protocol
    - Generates type-safe, transport independent Msg passing API
    - Comm. safety guaranteed by using API+standard Go type checking
- Evaluation: Framework is expressive, minimal runtime overhead

Omitted details

Projection, role inference, well-formedness check

→ Decidable!

Linearity

- Ensures a session runs from start to finish (no early termination)
- Channels are not re-used

→ Simple runtime check; but could be static

Error handling

- Idiomatic Go style -- natural to Go developers

Go runtime optimisations

- Many lessons learned (ask me about it!)