Imperial College London

Scalable session programming for heterogeneous high-performance systems

Nicholas Ng Nobuko Yoshida Wayne Luk
Imperial College London

September 23, 2013

Outline

Motivation

- Parallel architectures
  - Efficient use of hardware resources
  - eg. Multicore processors, computer clusters
  - Difficult to program (correctly)
- Most common MPI error [Intel survey, SE-HPCS'05]
  - Communication mismatch (send-receive)
  - Communication deadlocks

Contributions

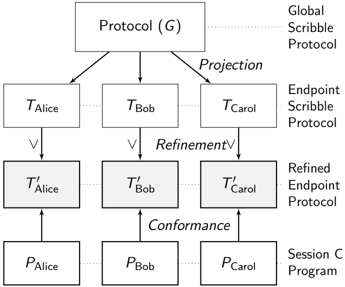
Session C	Type checking for session primitives
Pabble	Session-directed MPI code generation

- Two approaches to session-based parallel programming
  - Communication safety
  - Deadlock freedom
- Expressing scalable communication topologies as sessions
  - i.e. in our session-based protocol language

Imperial College London

Approach 1 - Session C programming

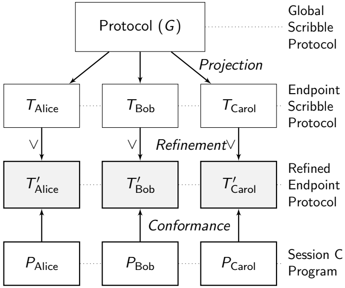
- Top-down approach
- Multiparty session types (MPST)
  - Communication: duality
  - Communication safety, deadlock freedom by typing



Imperial College London

Session C programming: Key reasoning

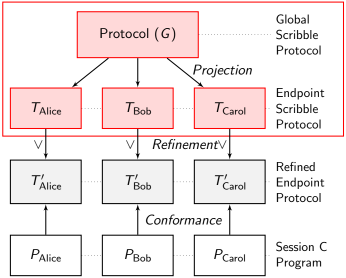
1. Design protocol in global view
2. Automatic *projection* to endpoint protocol, algorithm preserves safety
3. Write program according to endpoint protocol
4. Check program conforms to protocol
5. ⇒ Safe program by design



Imperial College London

Scribble protocol specification language

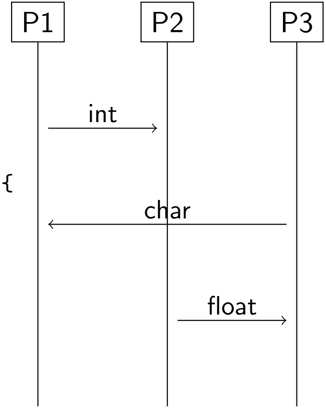
- Communication protocols
- Interaction by message passing
- Captures protocol control-flow
- Developed with industry
  - Red Hat
  - Cognizant
  - Ocean Observatories Initiative
- More details in tomorrow's demo session



Imperial College London

Scribble protocol specification language: Example

```
/* Global protocol */
protocol Simple
  (role P1, role P2, role P3) {
  int   from P1 to P2;
  char  from P3 to P1;
  float from P2 to P3
}
```

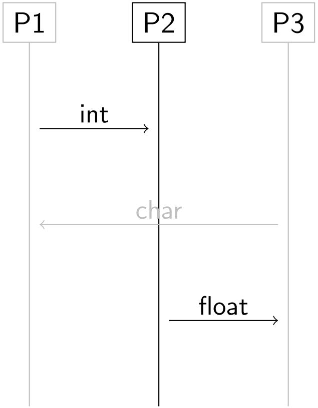


Imperial College London

Scribble protocol specification language: Example

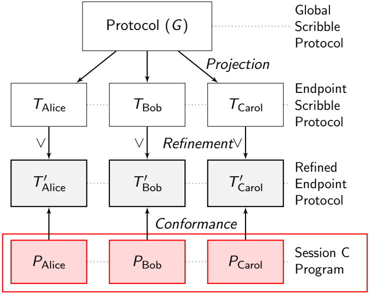
```
/* Endpoint protocol for P2 */
protocol Simple at P2
  (role P1, role P3) {
  int   from P1;

  float to P3;
}
```



Session C runtime

- Message passing API
- Built on 0MQ
  - Fast P2P communication
  - Lightweight
- Designed to be simple
  - Resembles Scribble
  - Some collective ops support



Imperial College London

Session C runtime: Examples

Iteration and message passing

Scribble

```
rec X {
  (int) to A;
  continue X;
}
```

```
rec Y {
  (int) from B;
  continue Y;
}
```

API (simple conditional)

```
while (i<3) {
  int val = 42;
  send_int(&val, 1, A);
}
```

```
while (i<3) {
  int val; recv_int(&val, 1, B);
}
```

Session C runtime: Examples

Directed choice

Scribble

```
choice to B {
  LABEL0(int) to B;
} or {
  LABEL1(int) to B; }
```

```
choice from A {
  LABEL0(int) from A;
} or {
  LABEL1(int) from A; }
```

API

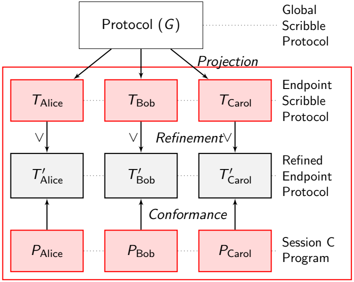
```
if (i<3) { // Choice from
  outbranch(B, LABEL0);
  send_int(B, 12);
} else {
  outbranch(B, LABEL1);
  send_char(B, 'A'); }
```

```
// Choice to
switch (inbranch(A, &label)) {
  case LABEL0:
    recv_int(A, &ival); break;
  case LABEL1:
    recv_char(A, &cval); break; }
```

Imperial College
London

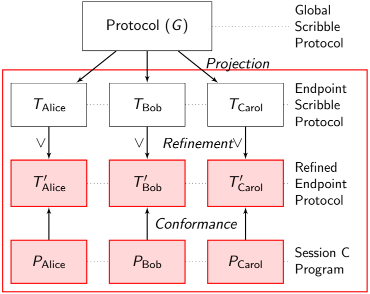
Session Type checking

- Static analyser
- Does source code conform to specification?
- Extract session type from code
  - Based on usage of API
  - Based on program flow control
- Compare w/ endpoint protocol



Session Type checking: Asynchronous optimisation

- Protocols designed safe
- Naive impl. inefficient
- Asynchronous impl.
  - Non-blocking send
  - Blocking receive
- Overlap send/recv operations
- Safety by async. subtyping [Mostrous et al., ESOP'09]



Summary (1/2): Session C programming framework

- Approach: Safety by type checking
- Protocol-based parallel programming framework
- Developer friendly Session Types as protocols
- Implemenation with custom API
- Guarantees communication safety, deadlock free by design

Approach 2: MPI Pabble Code generation approach

- Scaling: More practical parallel programming
- Message Passing Interface (MPI) is standard API
- Associate **Parameterised** MPST with MPI
  - Type representation (protocol)
    - Pabble: Parameterised Scribble
    - Scribble roles with indices
  - Type check/extraction from source code
    - Parameterised (dependent) type checking non-trivial
    - MPI deductive verification
      Related: next talk this session
- Our solution: Code generation from Pabble protocols

Writing a parallel pipeline in Scribble

```
global protocol Ring(role Worker1, role Worker2,
   role Worker3, role Worker4) {
  rec LOOP {
    Data(int) from Worker1 to Worker2 ;
    Data(int) from Worker2 to Worker3 ;
    Data(int) from Worker3 to Worker4 ;
    Data(int) from Worker4 to Worker1 ;
    continue LOOP;
  }
}
```

Pabble: Parameterised Scribble

- **Pa**rameterised Scri**bble** extension
- Role parameterisation by indices
- Grouping: Single endpoint protocol for parameterised roles
- Parametric extension of Scribble
  - `foreach`, recursion with loop index binding
  - `if`, conditional execution (multiple roles in single endpoint)
  - Role index calculation, design based on [Concurrency: state models and Java programs, Magee and Kramer, 2006]
- Scalable: Supports unbounded number of roles (for some cases)

Indexed interaction statement

Global protocol

```
Data(int) from Worker[i:1..9] to Worker[i+1];
```

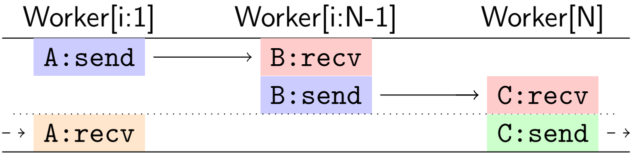
Endpoint protocol

- All `Worker`s share an endpoint protocol
- statements are executed conditionally (by index)

```
if Worker[i:2..10] Data(int) from Worker[i-1];
if Worker[i:1..9] Data(int) to Worker[i+1] ;
```

Example: Ring topology in Pabble

```
global protocol Ring(role Worker[1..N]) {
  rec LOOP {
    Data(int) from Worker[i:1..N-1] to Worker[i+1] ;
    Data(int) from Worker[N] to Worker[1] ;
    continue LOOP;
  }
}
```



Ring protocol: Worker endpoint

```
local protocol Ring at Worker[1..N](role Worker[1..N]) {
  rec LOOP {
    if Worker[i:2..N] Data(int) from Worker[i-1];
    if Worker[i:1..N-1] Data(int) to Worker[i+1];
    if Worker[1] Data(int) from Worker[N];
    if Worker[N] Data(int) to Worker[1];
    continue LOOP;
  }
}
```

MPI code generation

- Sessions and MPI: Similar program structure
  - Pabble also single-source multiple-endpoints
  - Parameterised role index = MPI ranks
- Pabble vs. core MPI primitives, e.g.
  - P2P: Send, Receive
  - Collective ops: Scatter, Gather, All to All

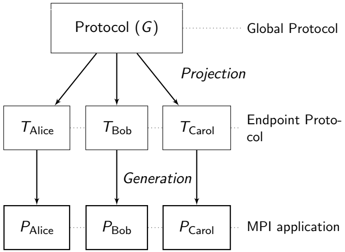
Ring protocol: Simplified MPI code

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &N);
while (1) { // rec LOOP
  // if Worker[i:2..N] Data(int) from Worker[i-1];
  if (2<=rank && rank<=N} MPI_Recv(.., MPI_INT, rank-1, Data, .. );
  // if Worker[i:1..N-1] Data(int) to Worker[i+1];
  if (1<=rank && rank<=N-1} MPI_Send(.., MPI_INT, rank+1, Data, ..
  // if Worker[1] Data(int) from Worker[N];
  if (rank==1} MPI_Recv(.., MPI_INT, N, Data, .. );
  // if Worker[N] Data(int) to Worker[1];
  if (rank==N} MPI_Recv(.., MPI_INT, 1, Data, .. );
}
MPI_Finalize();
```

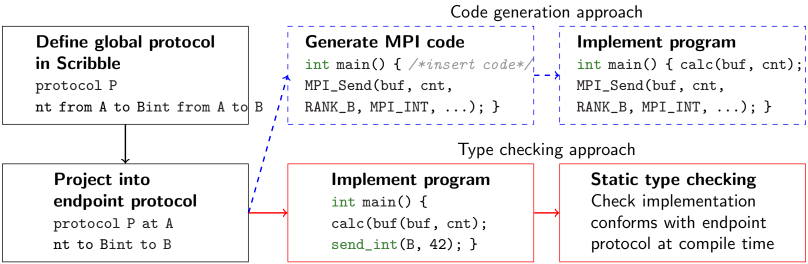
Imperial College London

Summary (2/2): MPI code generation from Pabble

- Approach: Safety by code generation
- Generate MPI backbone
  - Communication-correct
- Pabble indexed roles to rank
- Supports MPI collective ops



Conclusion: Session-based safe parallel programming



- Communication safety
- Deadlock free

Ongoing and future work

- Extract/verify Session Types from MPI
  - Huge engineering challenge
  - Can we infer global types from the endpoint MPI programs?
  - Ongoing collaboration with FCUL [EuroMPI'12, PLACES'13, BEAT2'13]
- Applying methodology in different environments
  - Software-Hardware communication (eg. FPGA, Maxeler)
  - Heterogeneous hardware (FPGA) code generation via AOP
- ...

Code generation approach

Define global protocol in Scribble

```
protocol P
nt from A to Bint from A to B
```

Generate MPI code

```
int main() { /*insert code*/
MPI_Send(buf, cnt,
RANK_B, MPI_INT, ...); }
```

Implement program

```
int main() { calc(buf, cnt);
MPI_Send(buf, cnt,
RANK_B, MPI_INT, ...); }
```

Type checking approach

Project into endpoint protocol

```
protocol P at A
nt to Bint to B
```

Implement program

```
int main() {
calc(buf(buf, cnt);
send_int(B, 42); }
```

Static type checking

Check implementation conforms with endpoint protocol at compile time