CHORAL

Object-Oriented Choreographic Programming

choral-lang.org

Saverio Giallorenzo

University of Bologna and INRIA

🌐 saveriogiallorenzo.com
📧 saverio.giallorenzo@gmail.com

Fabrizio Montesi

University of Southern Denmark

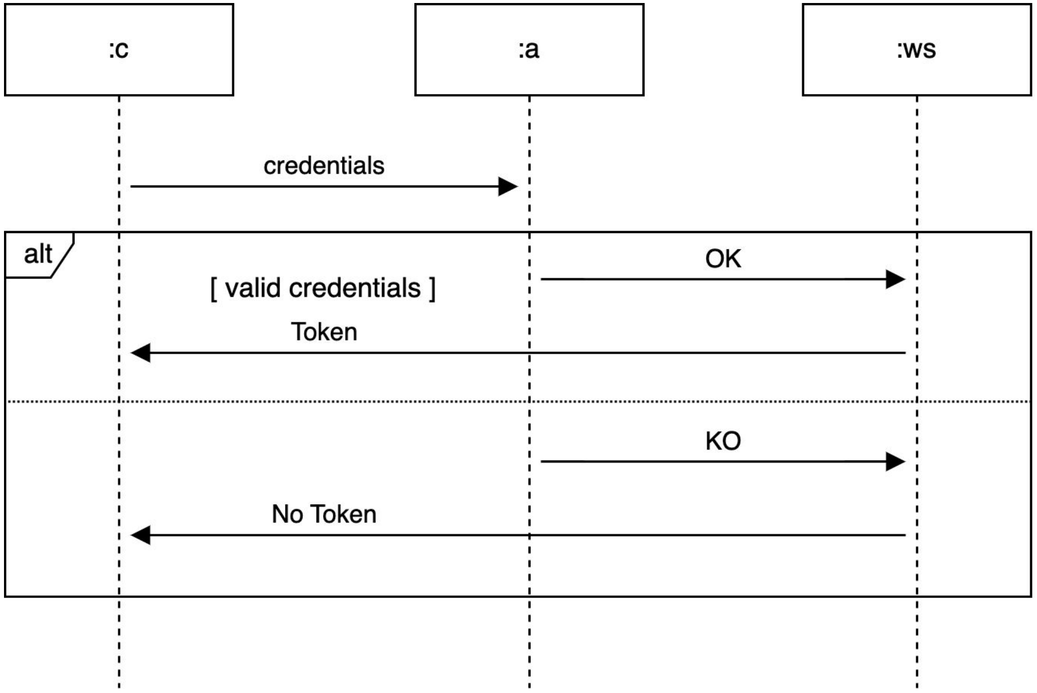
🌐 fabriziomontesi.com
📧 fmontesi@imada.sdu.dk
in linkedin.com/in/fmontesi
𝕏 x.com/famontesi

Marco Peressotti

University of Southern Denmark

🌐 marcoperessotti.com
📧 peressotti@imada.sdu.dk

A simple example from Single-Sign On (SSO):



Code for c

```
send credentials to a;
recv result from ws;
```

Code for a

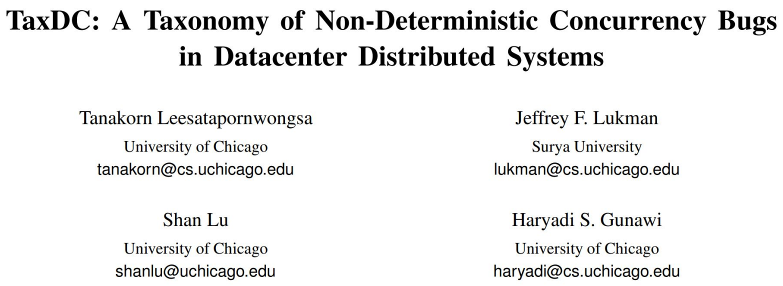
```
recv x from c;
if valid(x) {
      send OK to ws;
} else {
      send KO to ws;
}
```

Code for ws

```
recv decision from a;
switch(decision) {
case OK:
      send newToken() to c;
case KO:
      send NoToken to c;
}
```

Implementing Choreographies

✗ State explosion problem



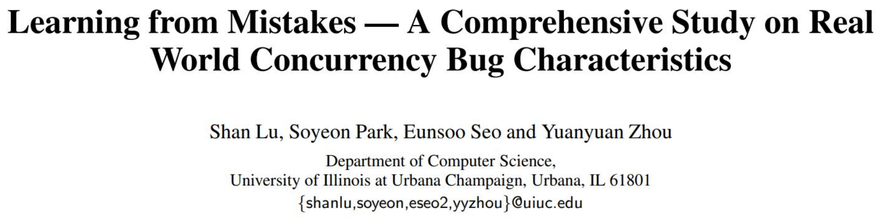
TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems

Tanakorn Leesatapornwongsa
University of Chicago
tanakorn@cs.uchicago.edu

Jeffrey F. Lukman
Surya University
lukman@cs.uchicago.edu

Shan Lu
University of Chicago
shanlu@uchicago.edu

Haryadi S. Gunawi
University of Chicago
haryadi@cs.uchicago.edu

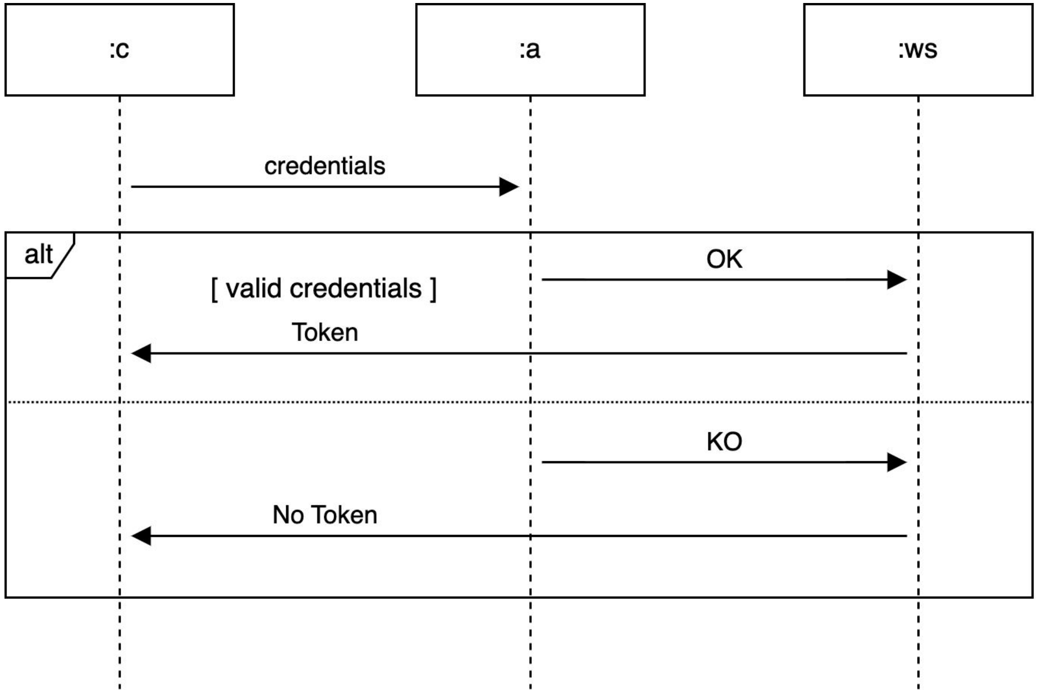


Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics

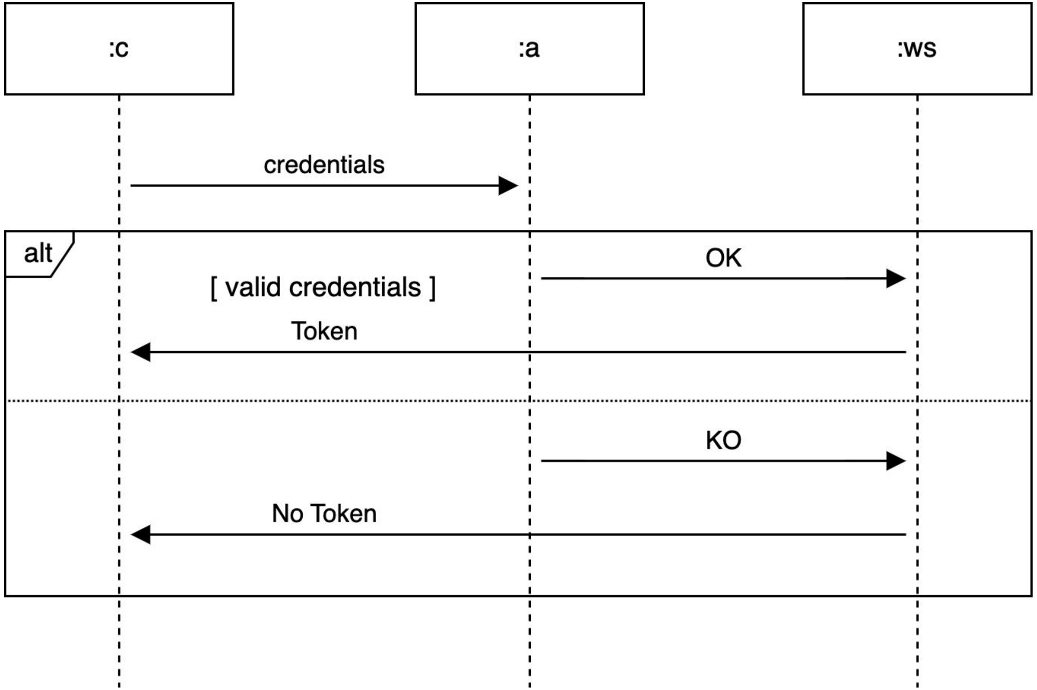
Shan Lu, Soyeon Park, Eunsoo Seo and Yuanyuan Zhou
Department of Computer Science,
University of Illinois at Urbana Champaign, Urbana, IL 61801
{shanlu,soyeon,eseo2,yyzhou}@uiuc.edu

Even expert programmers easily make mistakes!

Choreographic Programming



Choreographic Programming



Choreography

```
c.credentials -> a.x;
if a.valid(x) {
      a.OK -> ws.decision;
      ws.newToken() -> c.result
} else {
      a.KO -> ws.decision;
      ws.NoToken -> c.result
}
```

Choreographic Programming

Choreography

```
c.credentials -> a.x;
if a.valid(x) {
    a.OK -> ws.decision;
    ws.newToken() -> c.result
} else {
    a.KO -> ws.decision;
    ws.NoToken -> c.result
}
```

Choreographic Programming

Choreography

```
c.credentials -> a.x;
if a.valid(x) {
    a.OK -> ws.decision;
    ws.newToken() -> c.result
} else {
    a.KO -> ws.decision;
    ws.NoToken -> c.result
}
```

Choreographic Programming

Choreography

```
c.credentials -> a.x;
if a.valid(x) {
    a.OK -> ws.decision;
    ws.newToken() -> c.result
} else {
    a.KO -> ws.decision;
    ws.NoToken -> c.result
}
```

Choreographic Programming

Choreography

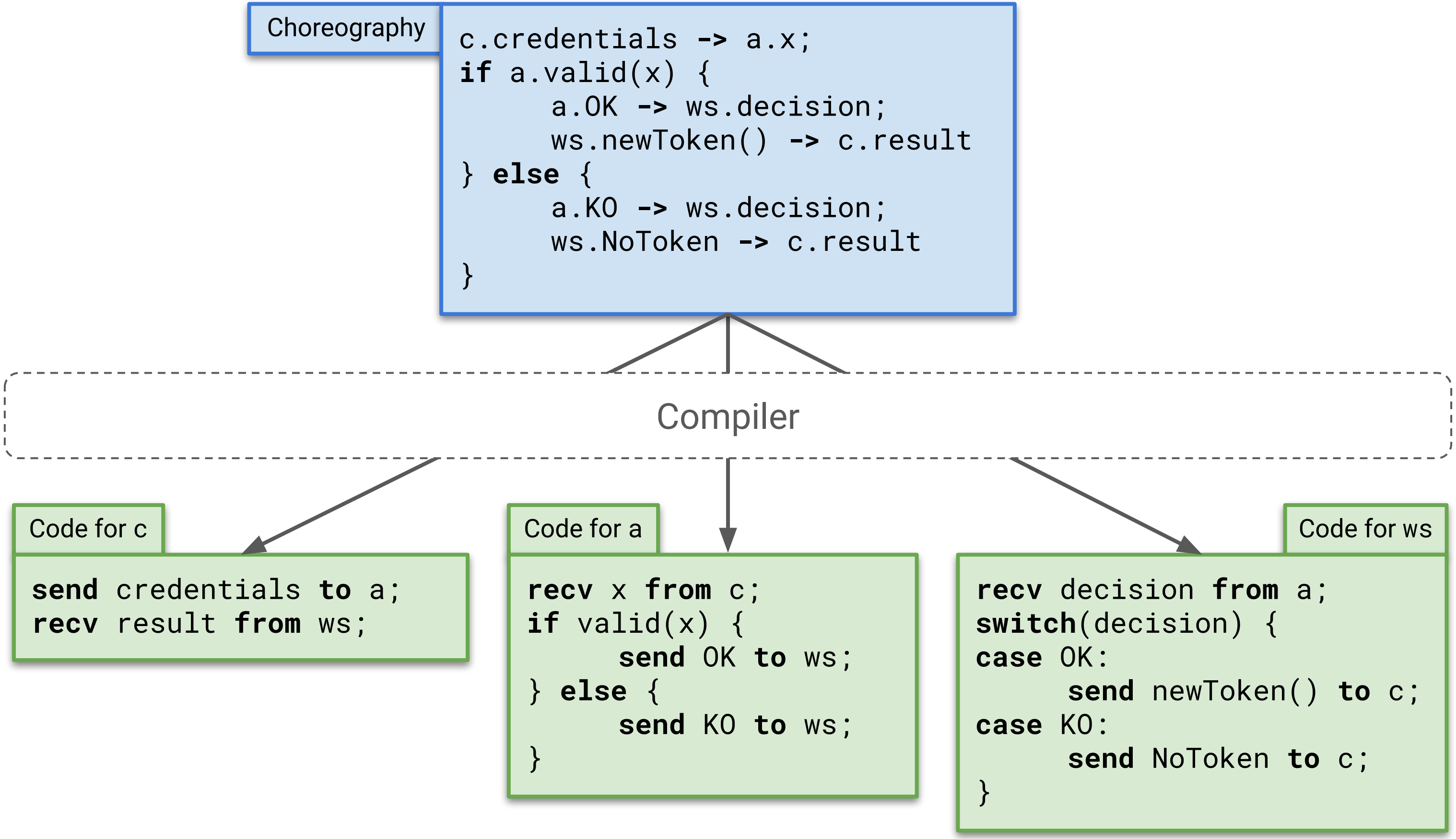
```
c.credentials -> a.x;
if a.valid(x) {
    a.OK -> ws.decision;
    ws.newToken() -> c.result
} else {
    a.KO -> ws.decision;
    ws.NoToken -> c.result
}
```

Choreographic Programming

Choreography

```
c.credentials -> a.x;
if a.valid(x) {
    a.OK -> ws.decision;
    ws.newToken() -> c.result
} else {
    a.KO -> ws.decision;
    ws.NoToken -> c.result
}
```

Choreographic Programming



Choreography

```
c.credentials -> a.x;
if a.valid(x) {
      a.OK -> ws.decision;
      ws.newToken() -> c.result
} else {
      a.KO -> ws.decision;
      ws.NoToken -> c.result
}
```

Compiler

Code for c

```
send credentials to a;
recv result from ws;
```

Code for a

```
recv x from c;
if valid(x) {
      send OK to ws;
} else {
      send KO to ws;
}
```

Code for ws

```
recv decision from a;
switch(decision) {
case OK:
      send newToken() to c;
case KO:
      send NoToken to c;
}
```

Properties of Choreographic Programming

Choreography Compliance

The system behaves as prescribed by the choreography.

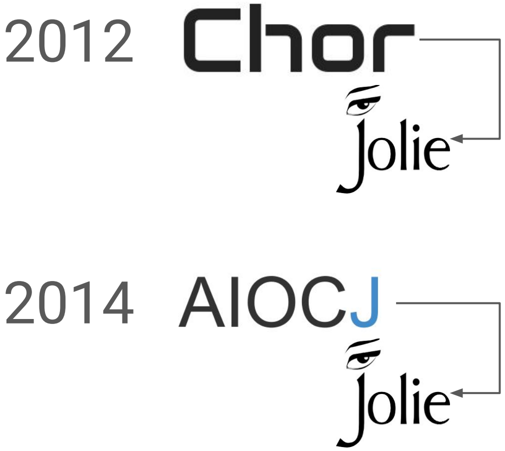
Communication Safety

Components do not perform incompatible actions.

Message Deadlock-Freedom

The system can always progress as a whole.

State of the Art



2012 Chor

Jolie

2014 AIOCJ

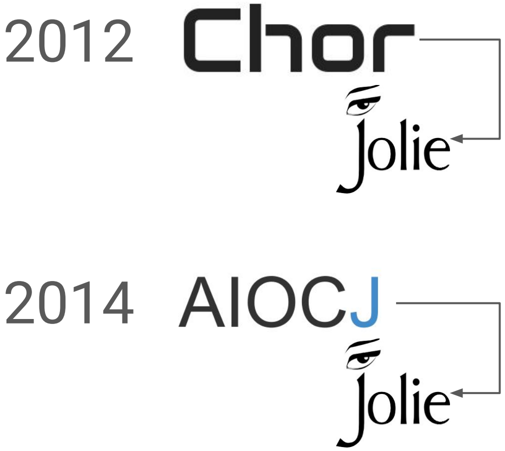
Jolie

First implemented choreographic programming language

- foundations from and language inspired by process calculi
- compilation to Jolie microservices

→ 'compiles to'

State of the Art



2012 Chor

Jolie

2014 AIOCJ

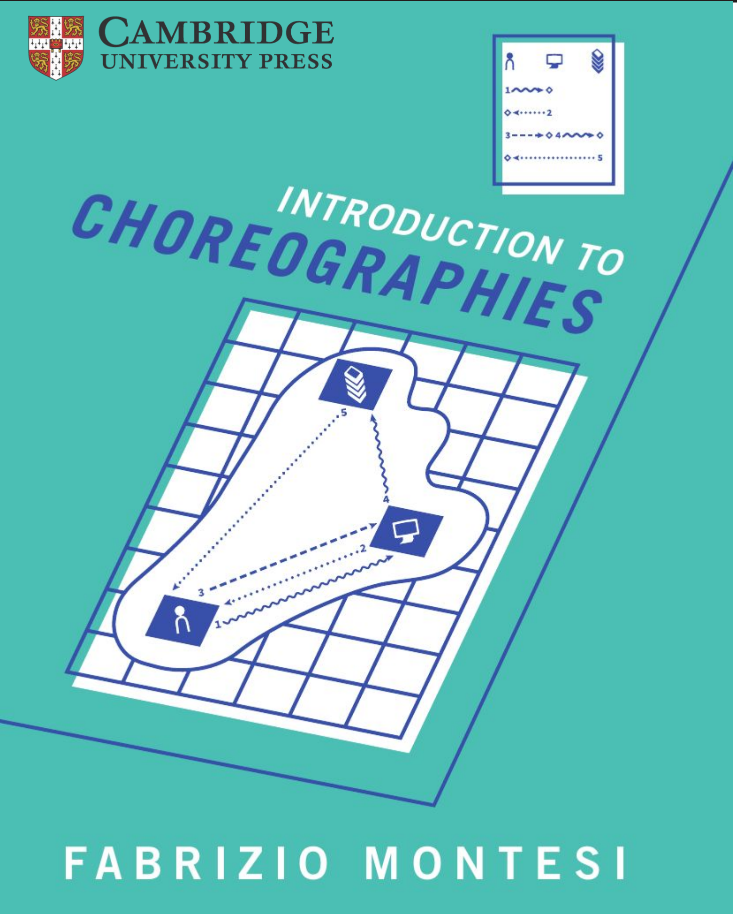
Jolie

First implemented choreographic programming language for dynamic adaptation (~modularity)

- foundations from and language inspired by process calculi
- compilation to Jolie framework for adaptation of microservice architectures

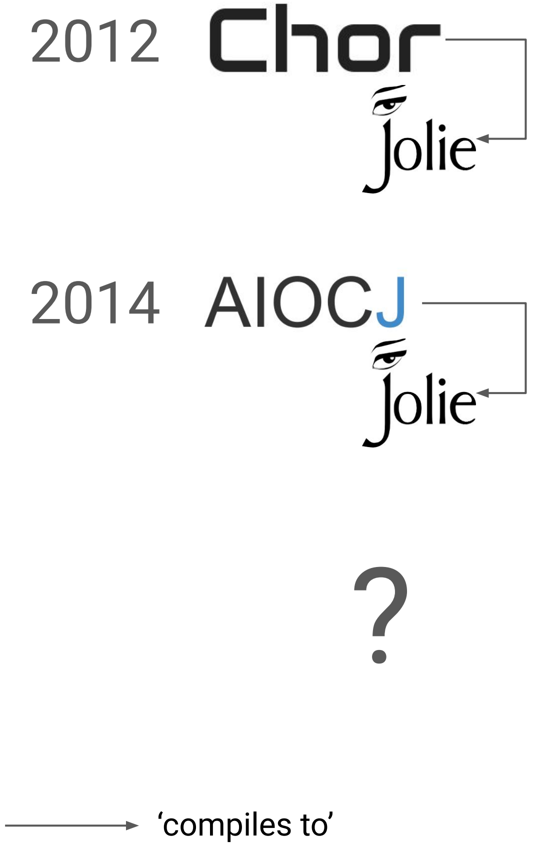
→ 'compiles to'

State of the Art



- ★ Well understood foundations.
- ★ How do we integrate choreographic programming with mainstream programming paradigms?
- ★ We need to tackle mainstream modular software development.

State of the Art



2012 Chor

Jolie

2014 AIOCJ

Jolie

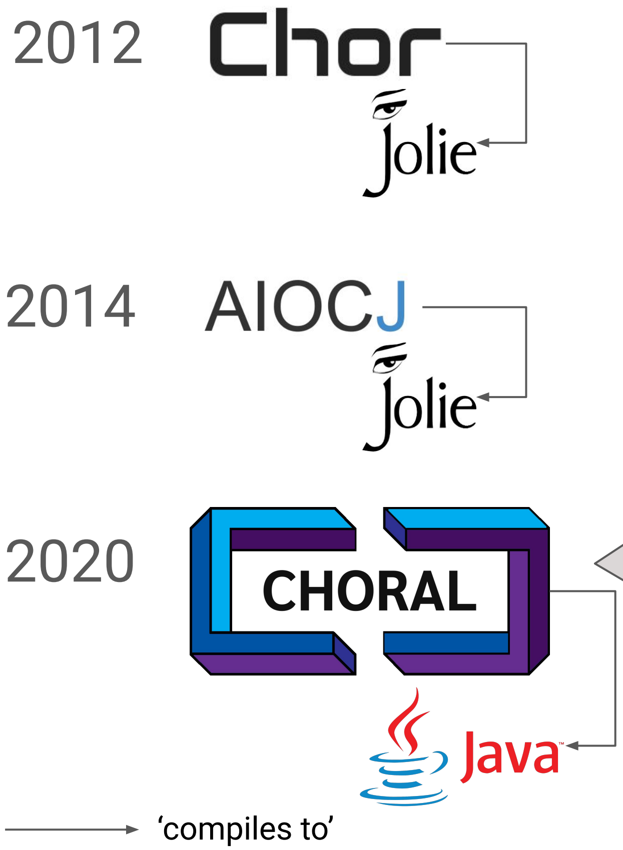
?

'compiles to'

How do we integrate choreographic programming with mainstream programming paradigms?

We need **modularity** (many ways to implement it) and **interoperability** with mainstream languages

This work



2012 Chor

Jolie

2014 AIOCJ

Jolie

2020 CHORAL

Java

'compiles to'

Choreographic programming for the real world.

- ★ **Modular and object-oriented**: we can now express protocols with abstraction, encapsulation, polymorphism, etc.
- ★ Fully **interoperable** with a mainstream language (Java).

This work

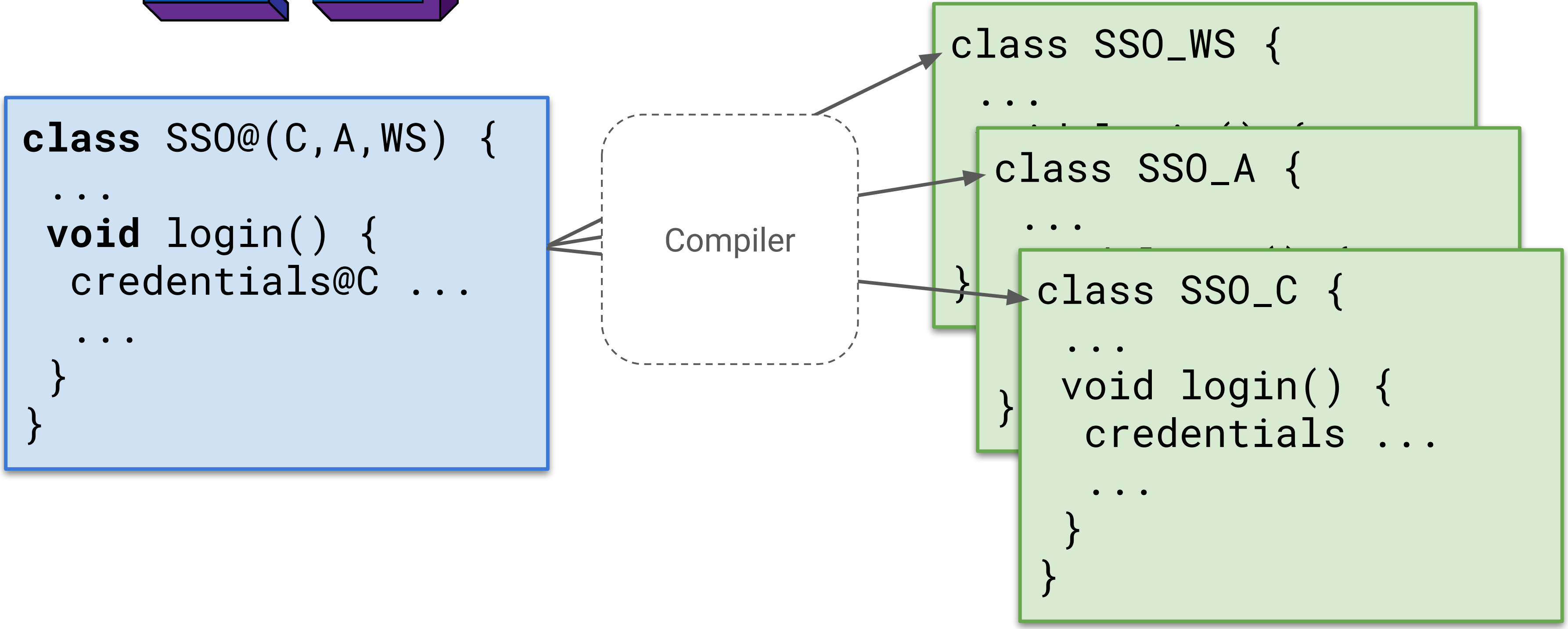
CHORAL

Choreographic programming for the real world.

★ **Modular and object-oriented**: we can now express protocols with abstraction, encapsulation, polymorphism, etc.

★ Fully **interoperable** with a mainstream language (Java).

This work



```
class SSO@(C,A,WS) {
 ...
 void login() {
  credentials@C ...
  ...
 }
}
```

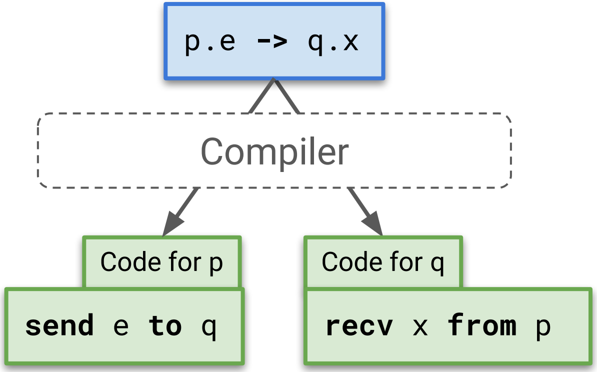
```
class SSO_WS {
 ...
}
```

```
class SSO_A {
 ...
}
```

```
class SSO_C {
 ...
 void login() {
  credentials ...
  ...
 }
}
```

A Taste of Choral

Reinterpreting `A -> B` in OO



p.e -> q.x

Compiler

Code for p

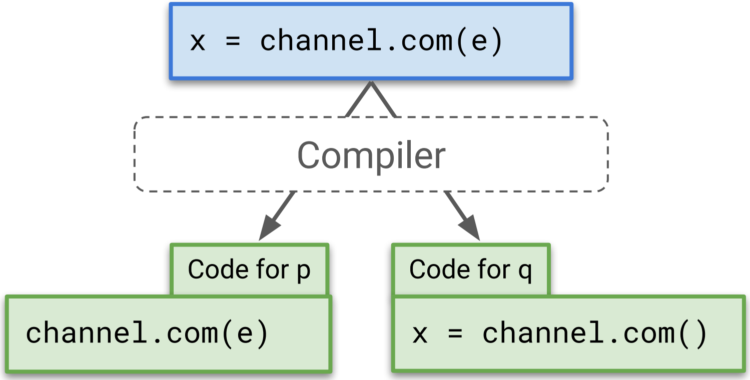
send e **to** q

Code for q

recv x **from** p

Traditionally:

- primitive statement
- syntactically combine send/recv



x = channel.com(e)

Compiler

Code for p

channel.com(e)

Code for q

x = channel.com()

Choral

- method invocation
- implementation is not fixed by the language
- data placement is tracked by types

Reinterpreting `A -> B` in OO

- Types track data placement
  - int@A for “an integer at A”
- A channel from A to B can be any object that offers a method with a signature like

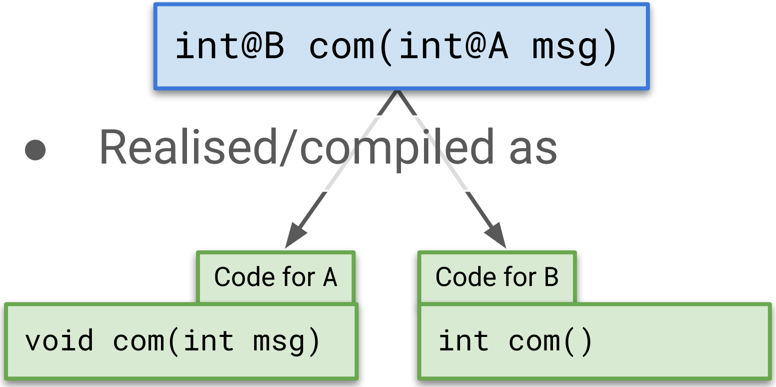
```
int@B com(int@A msg)
```

Reinterpreting `A -> B` in OO

- Types track data placement
  - int@A for “an integer at A”
- A channel from A to B can be any object that offers a method with a signature like

```
int@B com(int@A msg)
```

- Realised/compiled as



Code for A

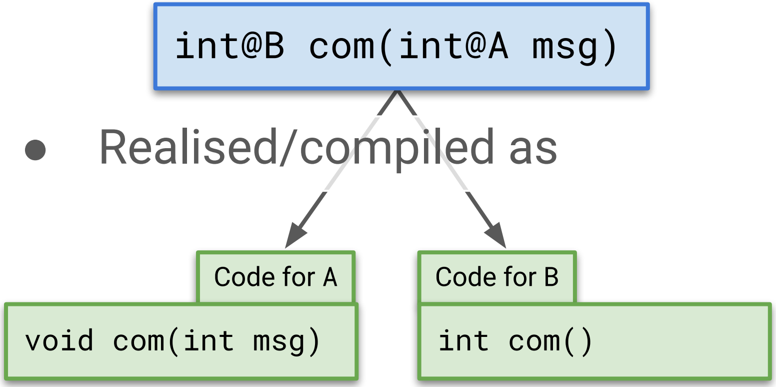
```
void com(int msg)
```

Code for B

```
int com()
```

Reinterpreting `A -> B` in OO

- Types track data placement
  - int@A for “an integer at A”
- A channel from A to B can be any object that offers a method with a signature like
- Realised/compiled as



```
int@B com(int@A msg)
```

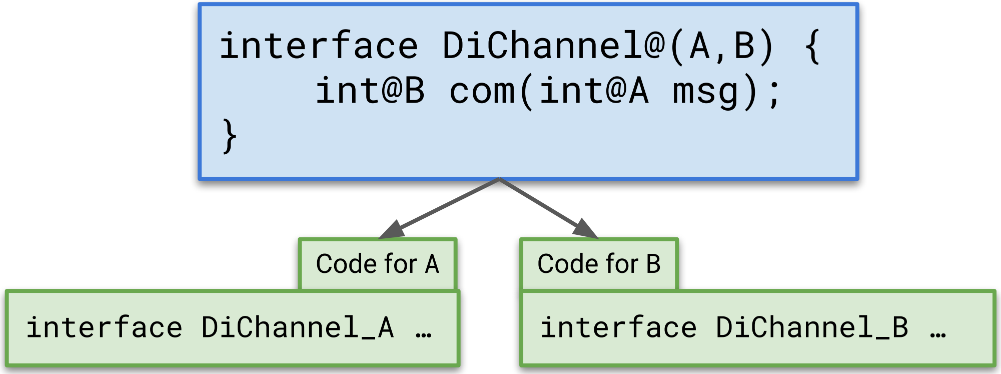
Code for A

```
void com(int msg)
```

Code for B

```
int com()
```

- Any such object must be distributed across A and B



```
interface DiChannel@(A,B) {
    int@B com(int@A msg);
}
```

Code for A

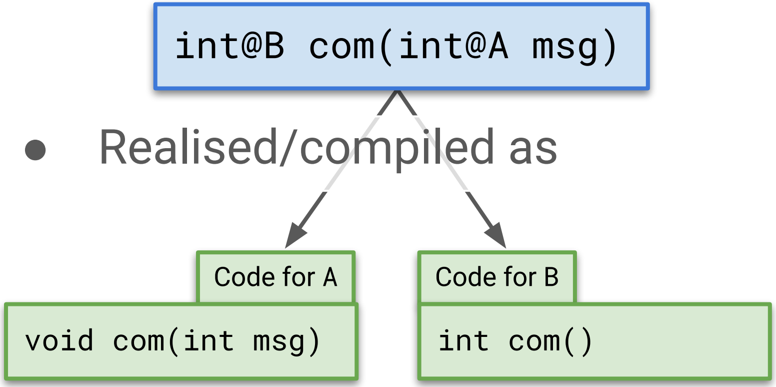
```
interface DiChannel_A …
```

Code for B

```
interface DiChannel_B …
```

Reinterpreting `A -> B` in OO

- Types track data placement
  - int@A for “an integer at A”
- A channel from A to B can be any object that offers a method with a signature like



```
int@B com(int@A msg)
```

- Realised/compiled as

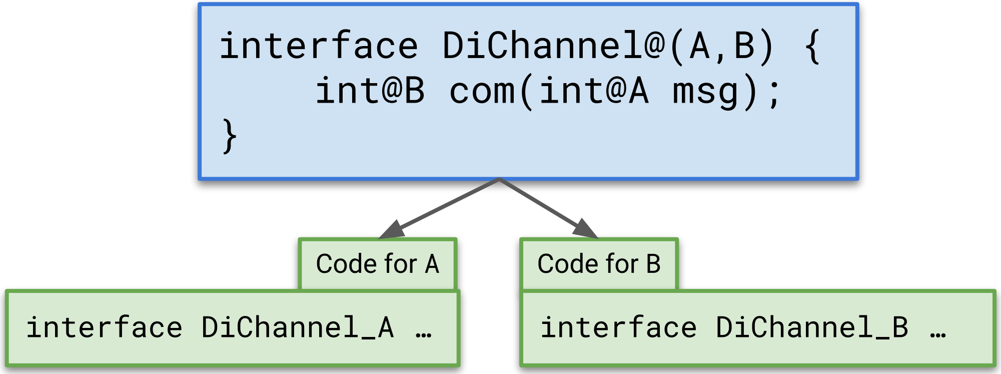
Code for A

```
void com(int msg)
```

Code for B

```
int com()
```

- Any such object must be distributed across A and B



```
interface DiChannel@(A,B) {
    int@B com(int@A msg);
}
```

Code for A

```
interface DiChannel_A …
```

Code for B

```
interface DiChannel_B …
```

- Just an ordinary interface in Choral:
  - no commitment to specific impl.
  - leverage OO principles

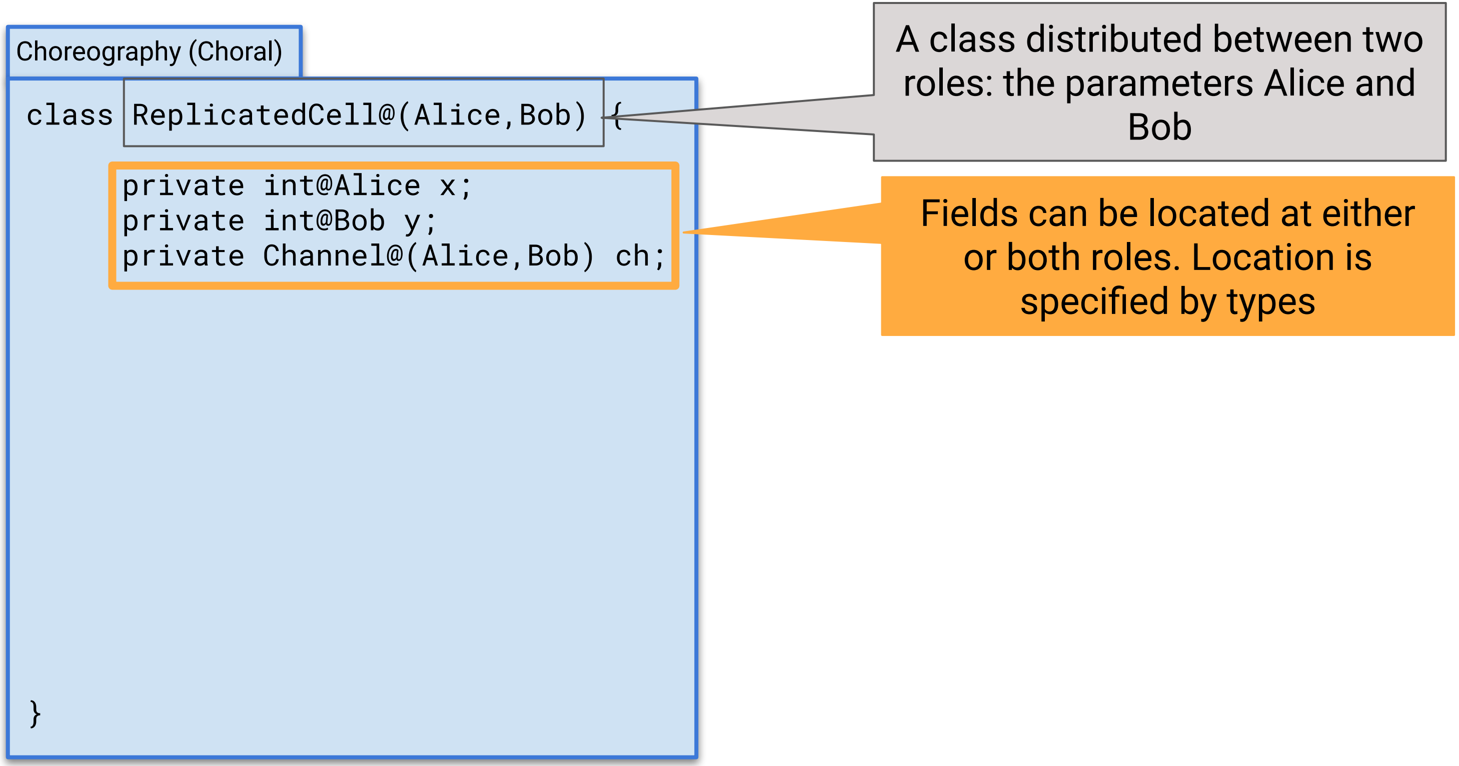
Distributed Data Structures

Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {






}
```

A class distributed between two roles: the parameters Alice and Bob

Distributed Data Structures



Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {
        private int@Alice x;
        private int@Bob y;
        private Channel@(Alice,Bob) ch;









}
```

A class distributed between two roles: the parameters Alice and Bob

Fields can be located at either or both roles. Location is specified by types

Distributed Data Structures

Choreography (Choral)

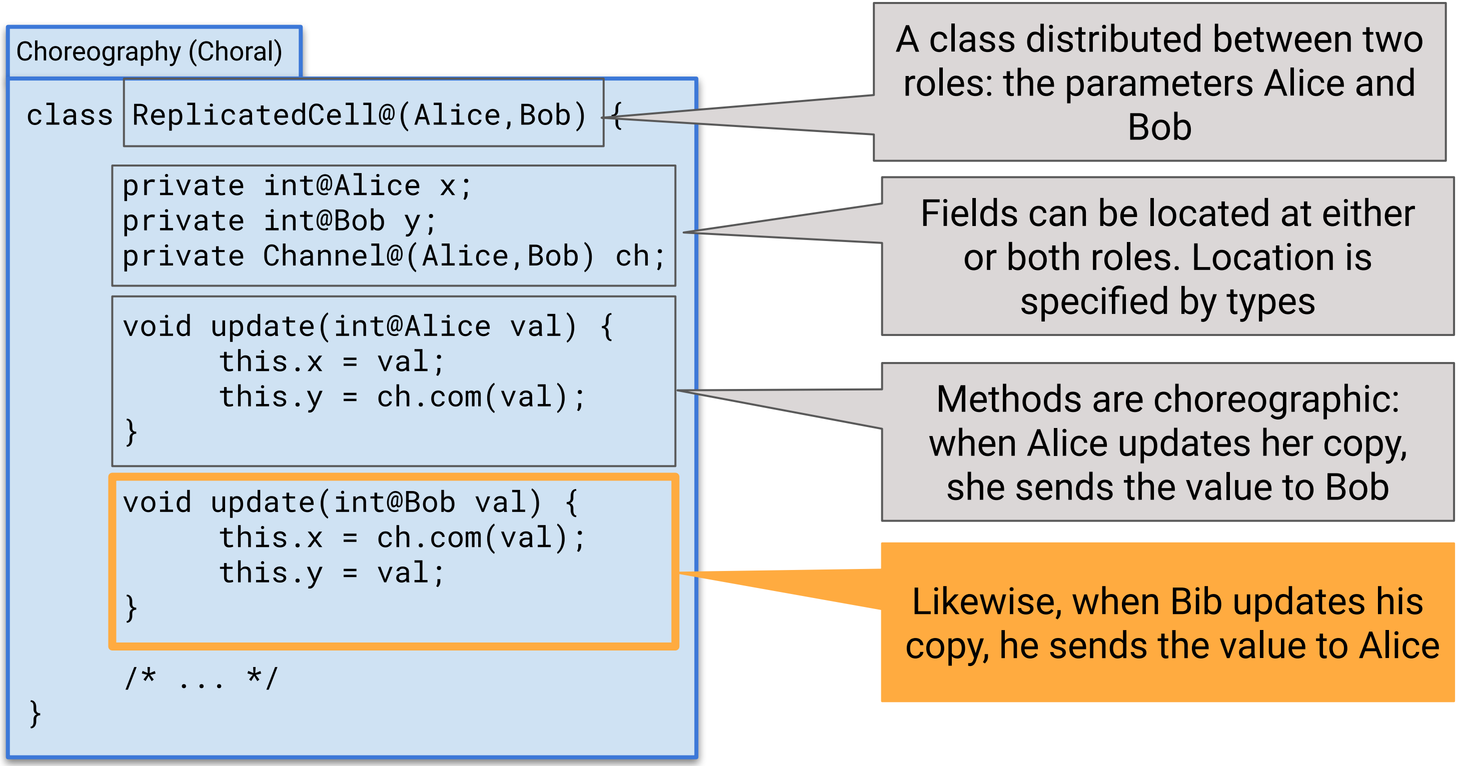
```
class ReplicatedCell@(Alice,Bob) {

      private int@Alice x;
      private int@Bob y;
      private Channel@(Alice,Bob) ch;

      void update(int@Alice val) {
            this.x = val;
            this.y = ch.com(val);
      }

      void update(int@Bob val) {
            this.x = ch.com(val);
            this.y = val;
      }

      /* ... */
}
```

A class distributed between two roles: the parameters Alice and Bob

Fields can be located at either or both roles. Location is specified by types

Methods are choreographic: when Alice updates her copy, she sends the value to Bob

Distributed Data Structures



Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {
          private int@Alice x;
          private int@Bob y;
          private Channel@(Alice,Bob) ch;

          void update(int@Alice val) {
                this.x = val;
                this.y = ch.com(val);
          }

          void update(int@Bob val) {
                this.x = ch.com(val);
                this.y = val;
          }

          /* ... */
}
```

A class distributed between two roles: the parameters Alice and Bob

Fields can be located at either or both roles. Location is specified by types

Methods are choreographic: when Alice updates her copy, she sends the value to Bob

Likewise, when Bib updates his copy, he sends the value to Alice

Distributed Data Structures

Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

     private int@Alice x;
     private int@Bob y;
     private Channel@(Alice,Bob) ch;

     void update(int@Alice val) {
          this.x = val;
          this.y = ch.com(val);
     }

     void update(int@Bob val) {
          this.x = ch.com(val);
          this.y = val;
     }

     /* ... */
}
```

Implementation for Alice (Java)

Distributed Data Structures

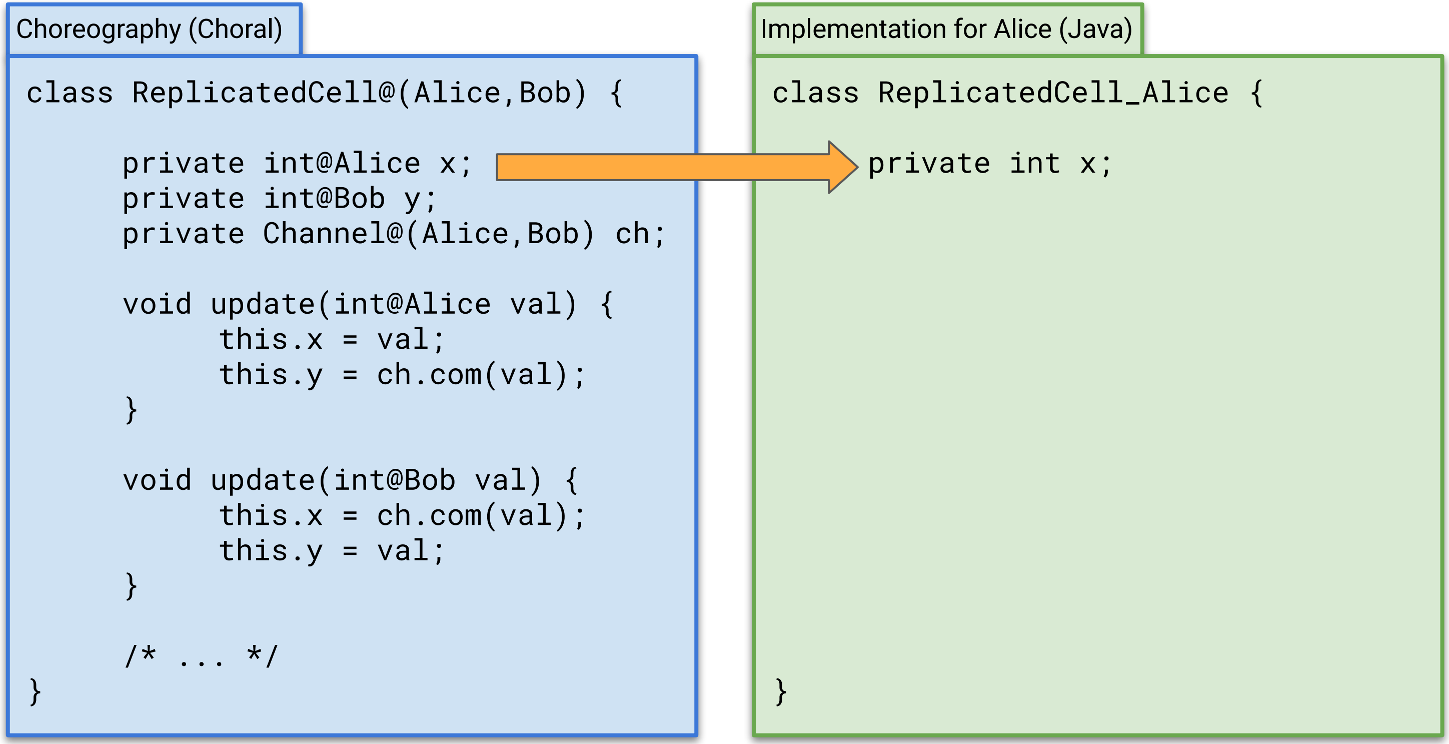
Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

     private int@Alice x;
     private int@Bob y;
     private Channel@(Alice,Bob) ch;

     void update(int@Alice val) {
          this.x = val;
          this.y = ch.com(val);
     }

     void update(int@Bob val) {
          this.x = ch.com(val);
          this.y = val;
     }

     /* ... */
}
```

Implementation for Alice (Java)

```
class ReplicatedCell_Alice {









}
```

Distributed Data Structures



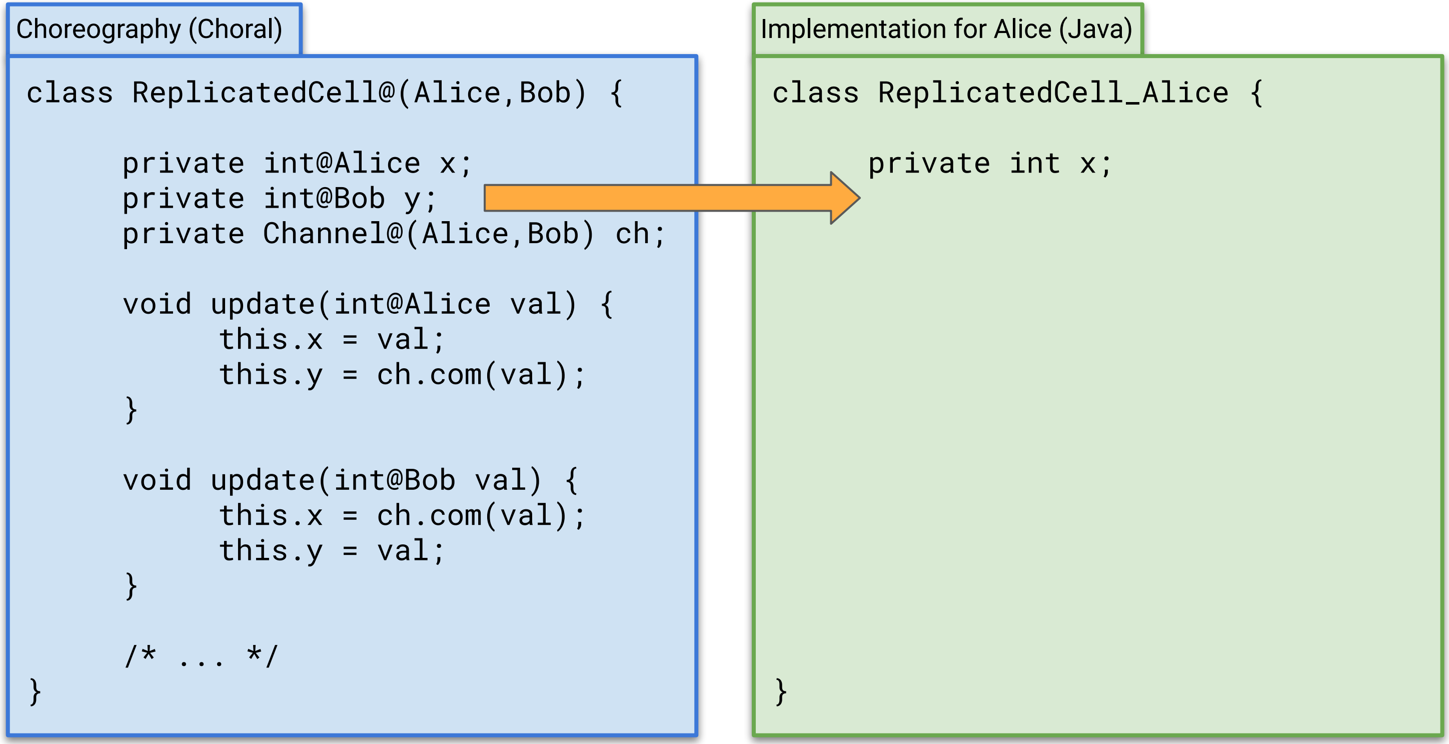
Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

     private int@Alice x;
     private int@Bob y;
     private Channel@(Alice,Bob) ch;

     void update(int@Alice val) {
          this.x = val;
          this.y = ch.com(val);
     }

     void update(int@Bob val) {
          this.x = ch.com(val);
          this.y = val;
     }

     /* ... */
}
```

Implementation for Alice (Java)

```
class ReplicatedCell_Alice {

     private int x;

}
```

Distributed Data Structures



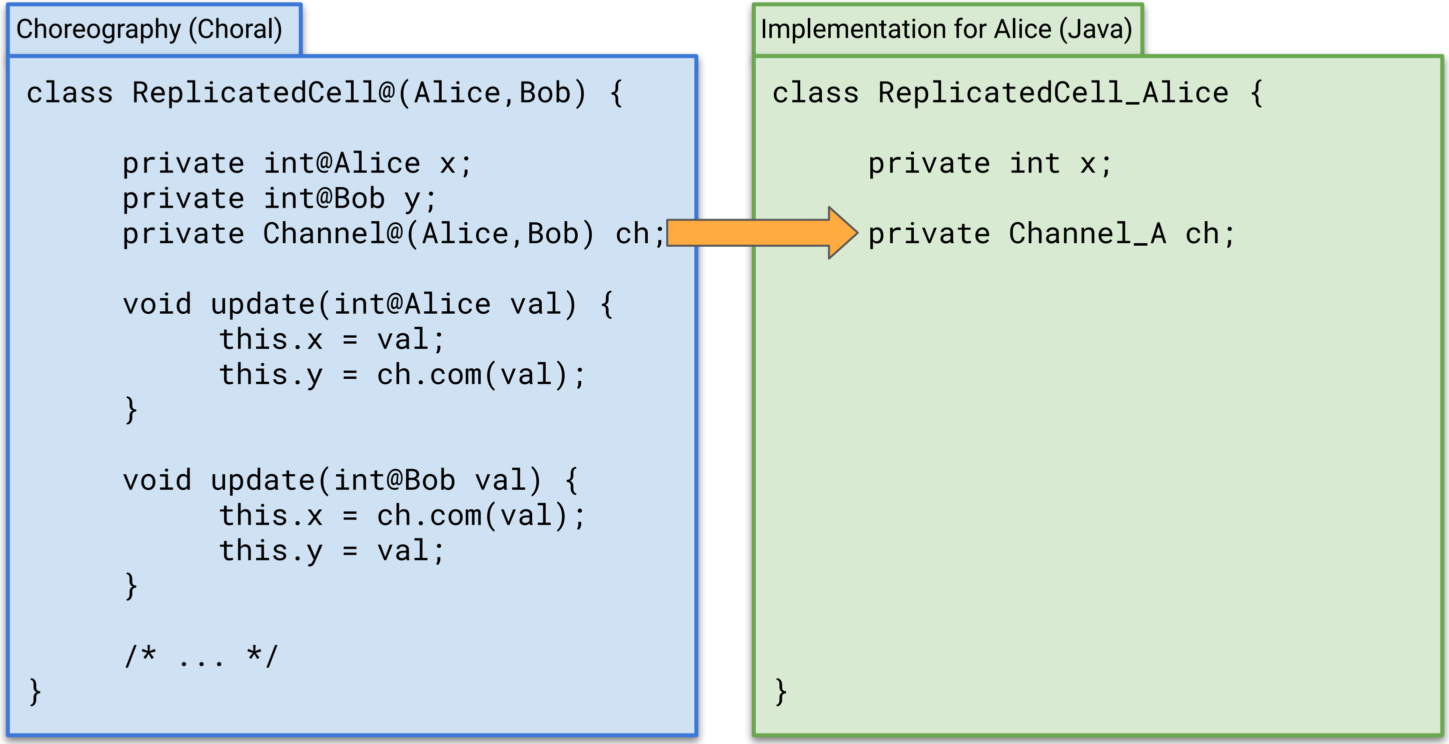
Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

	private int@Alice x;
	private int@Bob y;
	private Channel@(Alice,Bob) ch;

	void update(int@Alice val) {
		this.x = val;
		this.y = ch.com(val);
	}

	void update(int@Bob val) {
		this.x = ch.com(val);
		this.y = val;
	}

	/* ... */
}
```

Implementation for Alice (Java)

```
class ReplicatedCell_Alice {

	private int x;







}
```

Distributed Data Structures



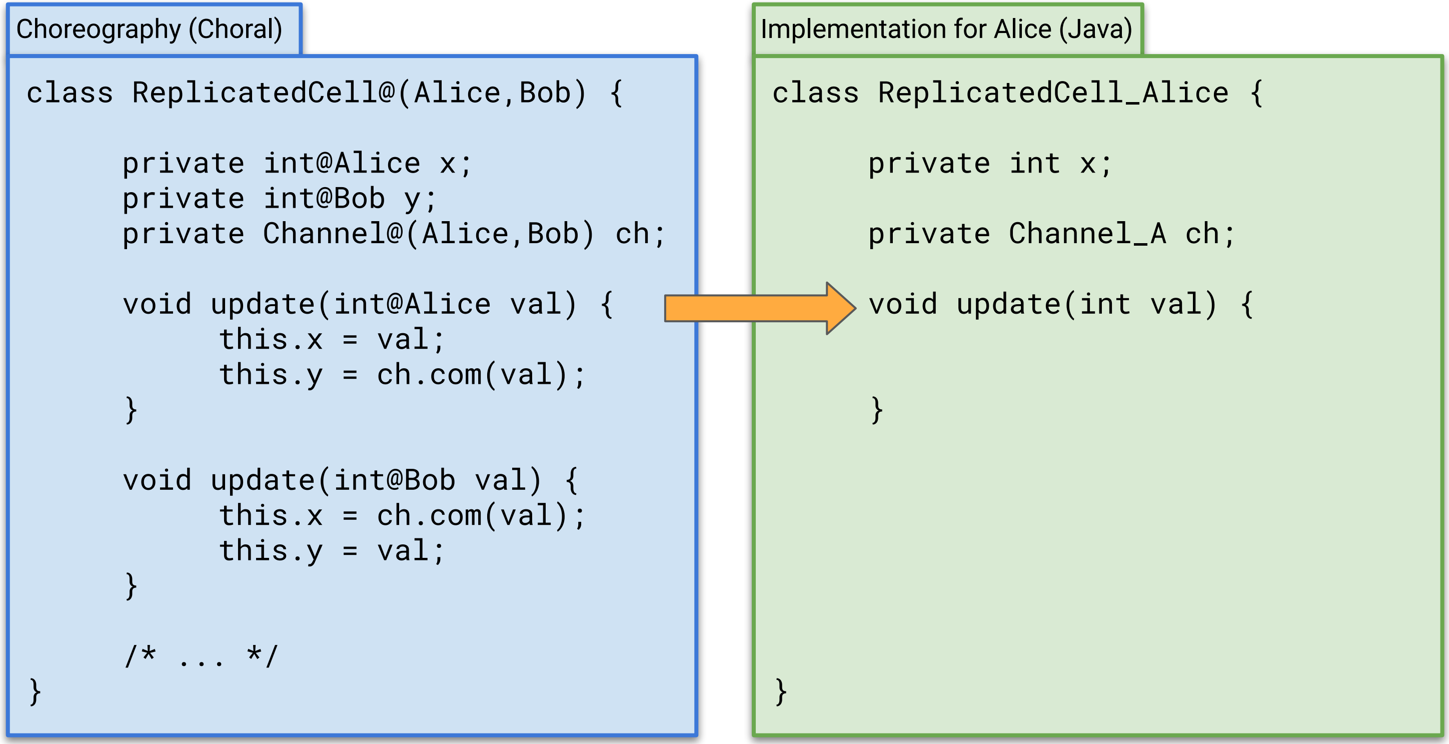
Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

     private int@Alice x;
     private int@Bob y;
     private Channel@(Alice,Bob) ch;

     void update(int@Alice val) {
          this.x = val;
          this.y = ch.com(val);
     }

     void update(int@Bob val) {
          this.x = ch.com(val);
          this.y = val;
     }

     /* ... */
}
```

Implementation for Alice (Java)

```
class ReplicatedCell_Alice {

     private int x;

     private Channel_A ch;






}
```

Distributed Data Structures



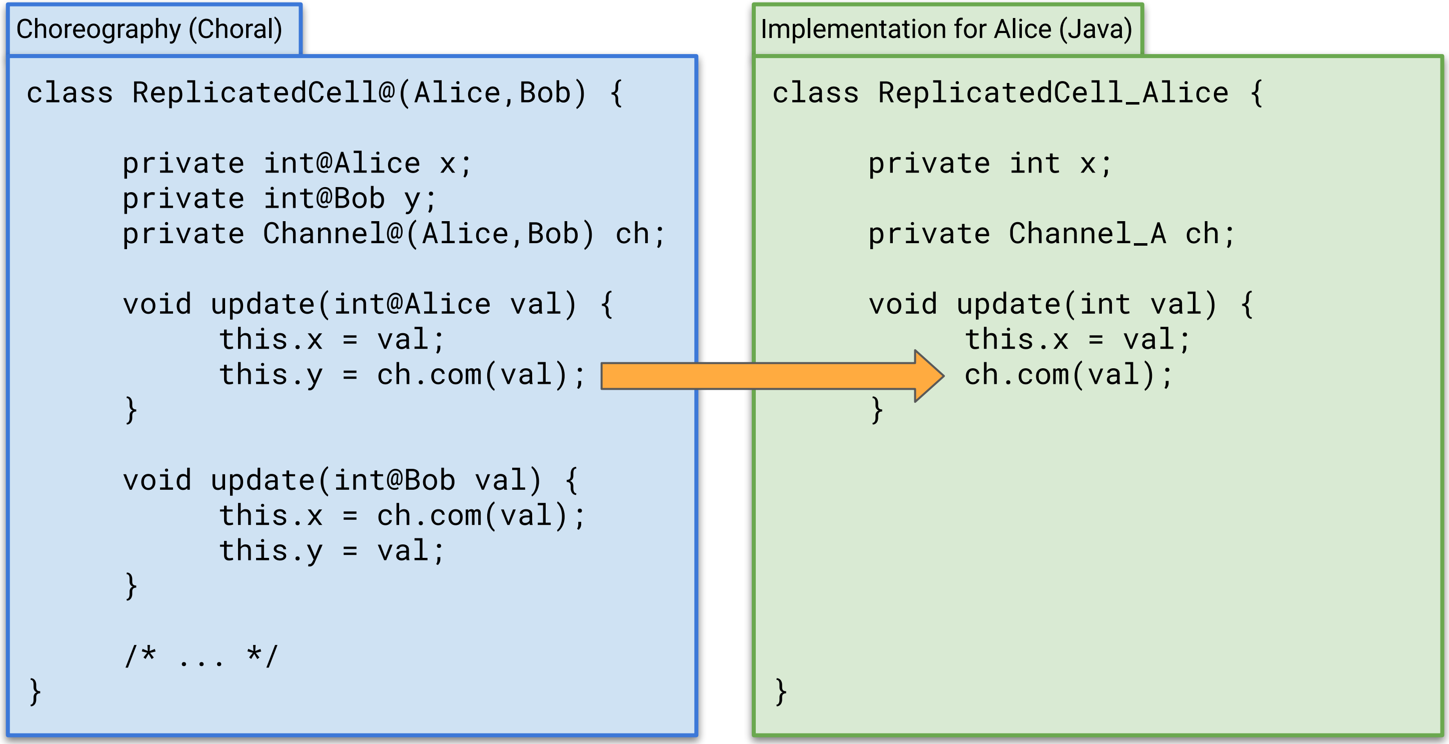
Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

    private int@Alice x;
    private int@Bob y;
    private Channel@(Alice,Bob) ch;

    void update(int@Alice val) {
        this.x = val;
        this.y = ch.com(val);
    }

    void update(int@Bob val) {
        this.x = ch.com(val);
        this.y = val;
    }

    /* ... */
}
```

Implementation for Alice (Java)

```
class ReplicatedCell_Alice {

    private int x;

    private Channel_A ch;

    void update(int val) {


    }







}
```

Distributed Data Structures



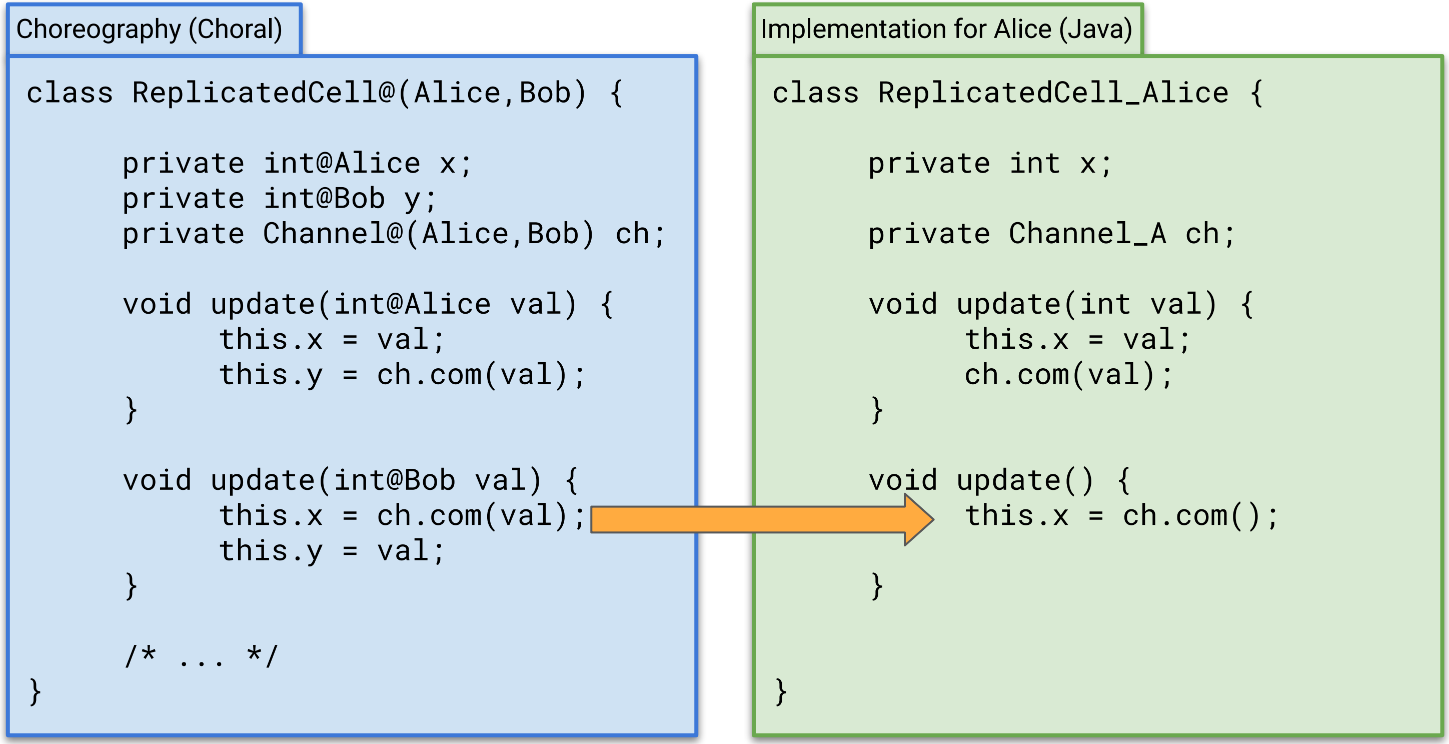
Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

    private int@Alice x;
    private int@Bob y;
    private Channel@(Alice,Bob) ch;

    void update(int@Alice val) {
        this.x = val;
        this.y = ch.com(val);
    }

    void update(int@Bob val) {
        this.x = ch.com(val);
        this.y = val;
    }

    /* ... */
}
```

Implementation for Alice (Java)

```
class ReplicatedCell_Alice {

    private int x;

    private Channel_A ch;

    void update(int val) {
        this.x = val;
        ch.com(val);
    }




}
```

Distributed Data Structures



Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

    private int@Alice x;
    private int@Bob y;
    private Channel@(Alice,Bob) ch;

    void update(int@Alice val) {
        this.x = val;
        this.y = ch.com(val);
    }

    void update(int@Bob val) {
        this.x = ch.com(val);
        this.y = val;
    }

    /* ... */
}
```

Implementation for Alice (Java)

```
class ReplicatedCell_Alice {

    private int x;

    private Channel_A ch;

    void update(int val) {
        this.x = val;
        ch.com(val);
    }

    void update() {
        this.x = ch.com();

    }


}
```

Distributed Data Structures

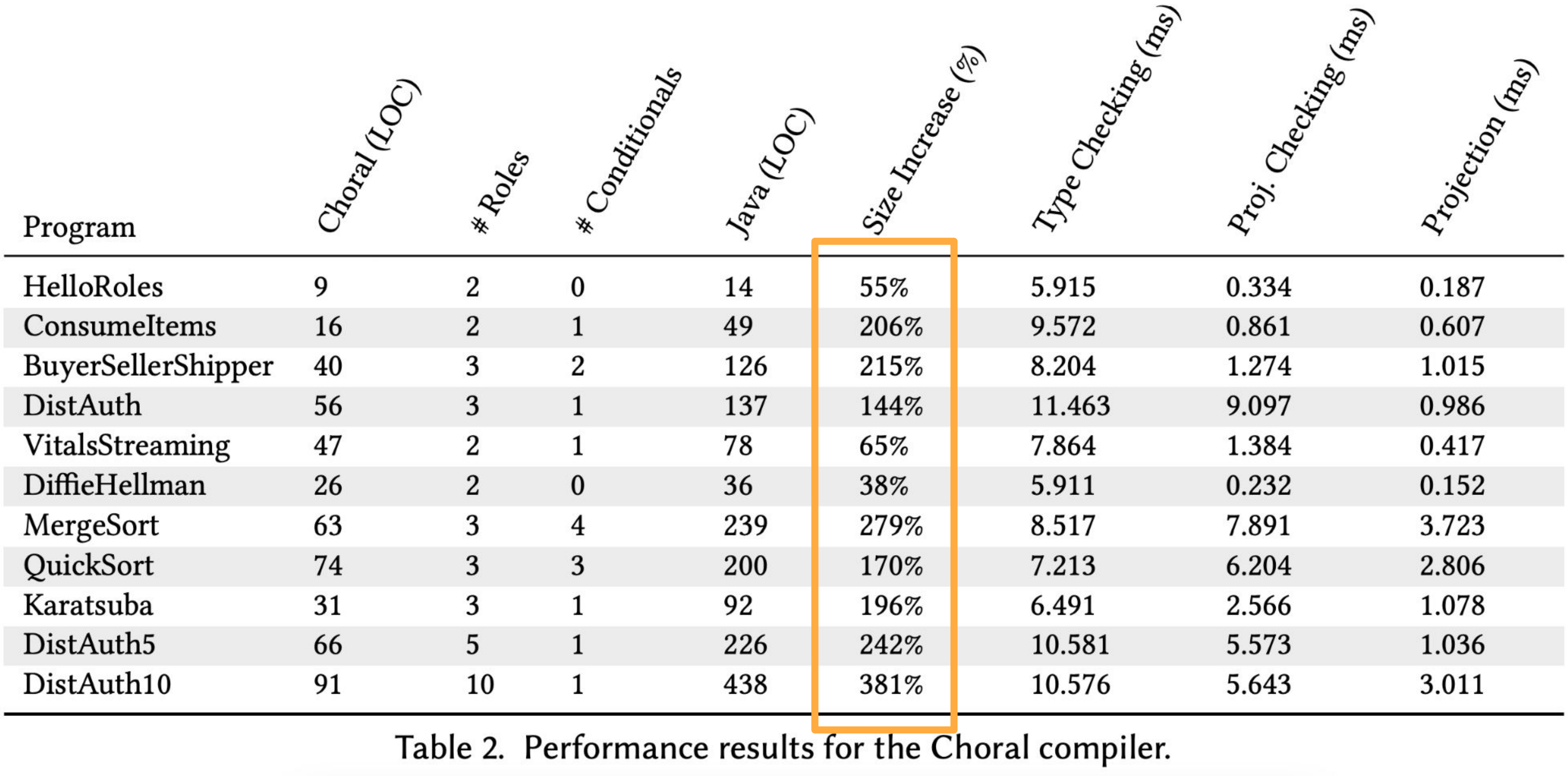
Choreography (Choral)

```
class ReplicatedCell@(Alice,Bob) {

      private int@Alice x;
      private int@Bob y;
      private Channel@(Alice,Bob) ch;

      void update(int@Alice val) {
            this.x = val;
            this.y = ch.com(val);
      }

      void update(int@Bob val) {
            this.x = ch.com(val);
            this.y = val;
      }

      /* ... */
}
```

Implementation for Alice (Java)

```
class ReplicatedCell_Alice {

      private int x;

      private Channel_A ch;

      void update(int val) {
            this.x = val;
            ch.com(val);
      }

      void update() {
            this.x = ch.com();

      }

      /* ... */
}
```

Evaluation: compiler performance



Program	Choral (LOC)	# Roles	# Conditionals	Java (LOC)	Size Increase (%)	Type Checking (ms)	Proj. Checking (ms)	Projection (ms)
HelloRoles	9	2	0	14	55%	5.915	0.334	0.187
ConsumeItems	16	2	1	49	206%	9.572	0.861	0.607
BuyerSellerShipper	40	3	2	126	215%	8.204	1.274	1.015
DistAuth	56	3	1	137	144%	11.463	9.097	0.986
VitalsStreaming	47	2	1	78	65%	7.864	1.384	0.417
DiffieHellman	26	2	0	36	38%	5.911	0.232	0.152
MergeSort	63	3	4	239	279%	8.517	7.891	3.723
QuickSort	74	3	3	200	170%	7.213	6.204	2.806
Karatsuba	31	3	1	92	196%	6.491	2.566	1.078
DistAuth5	66	5	1	226	242%	10.581	5.573	1.036
DistAuth10	91	10	1	438	381%	10.576	5.643	3.011

Table 2. Performance results for the Choral compiler.

Evaluation: Architecture Refactoring

- Considered an existing application
  - A reference implementation of an open source microblogging platform (Retwis)
  - Monolith implementation (JSP + Redis)

- Created a distributed version (monolith -> microservices)
  - Choral for programming interactions among distributed components
  - Reused the original logic and data structures
  - Drop-in replacement (same clients, same database)

CHORAL

- Modular and object-oriented
- Fully interoperable with a mainstream language (Java).
- Compilation is formally specified (*)
- Evaluation
  - Realistic algorithms and architectures
  - Comparison with Akka and Java (*)
- Development methodology (*)
- Testing framework (*)

(*) not in this talk, see paper

Thank you for listening!

Q&A