Choreographies: on Mainstream Tides

Saverio Giallorenzo

Università di Bologna (IT) INRIA (FR)

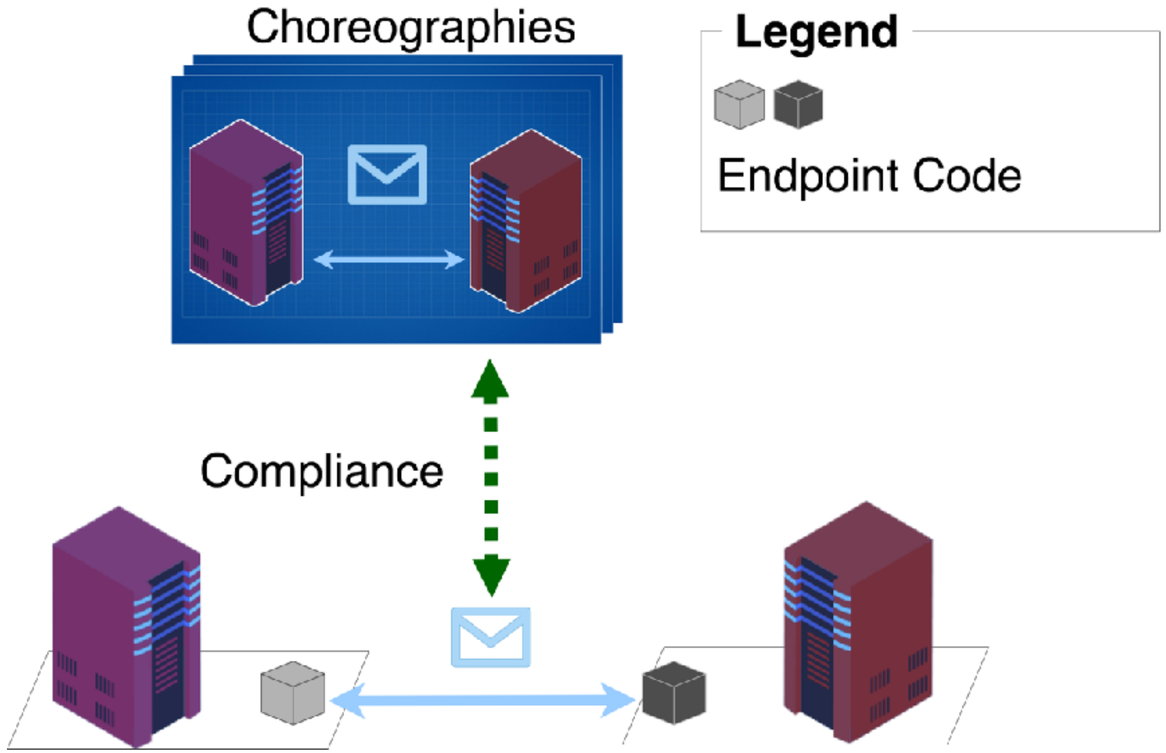
Choreographies

1. **User** knows $\{password\}$
2. **User** $\rightarrow$ **SSO** : $\{password\}$;
3. **SSO** generates $\{token\}$;
4. **SSO** $\rightarrow$ **Server** : $\{token\}$;
3. **Server** stores $\{token\}$;
5. **SSO** $\rightarrow$ **User** : $\{token\}; \epsilon$

Briais, Sébastien; Nestmann, Uwe (2007). "A formal semantics for protocol narrations". *Theoretical Computer Science*. Elsevier BV. **389** (3): 484–511.

Choreographies, compliance

1. **User** knows $\{password\}$
2. **User** $\rightarrow$ **SSO** : $\{password\}$;
3. **SSO** generates $\{token\}$;
4. **SSO** $\rightarrow$ **Server** : $\{token\}$;
3. **Server** stores $\{token\}$;
5. **SSO** $\rightarrow$ **User** : $\{token\}; \epsilon$



Choreographies, compliance

USR

1. know $\{password\}$
2. send to **SSO** : $\{password\}$;
5. receive from **SSO** : $\{token\}; \epsilon$

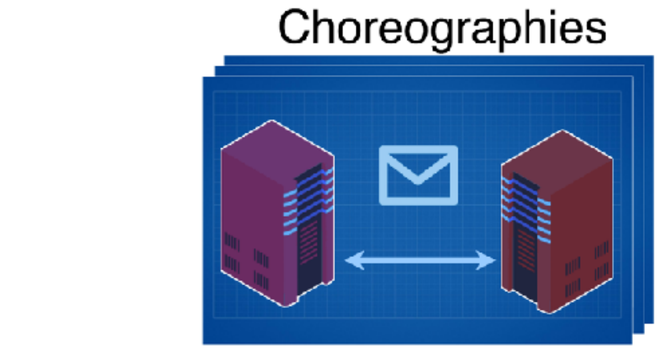
SSO

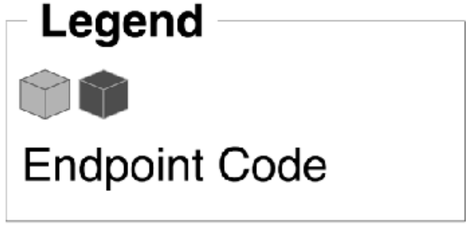
2. receive from **User** : $\{password\}$;
3. generate $\{token\}$;
4. send to **Server** : $\{token\}$;
5. send to **User** : $\{token\}; \epsilon$

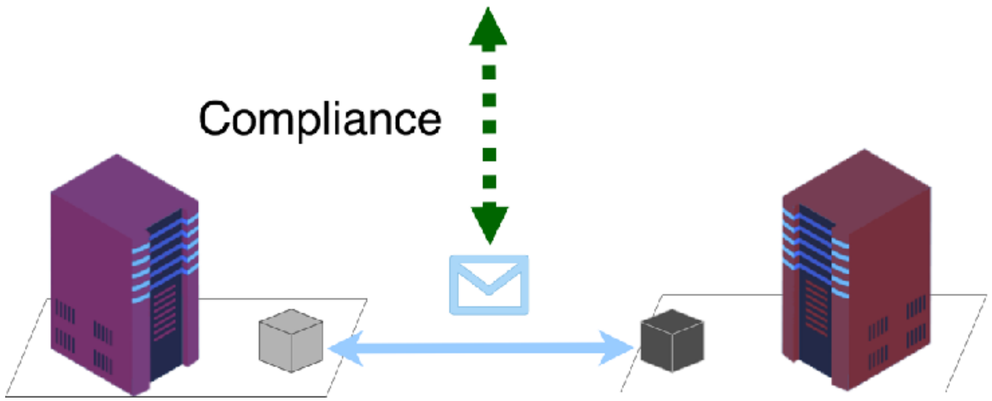
SRV

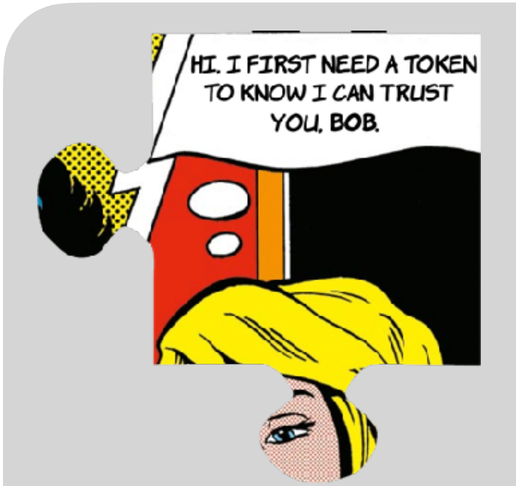
4. receive from **SSO** : $\{token\}$;
3. store $\{token\}; \epsilon$

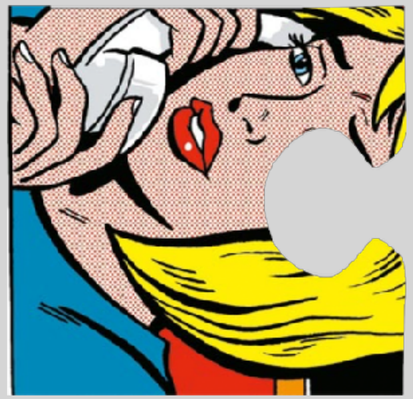
Choreographies, compliance







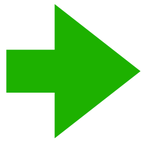




Choreographies, compliance

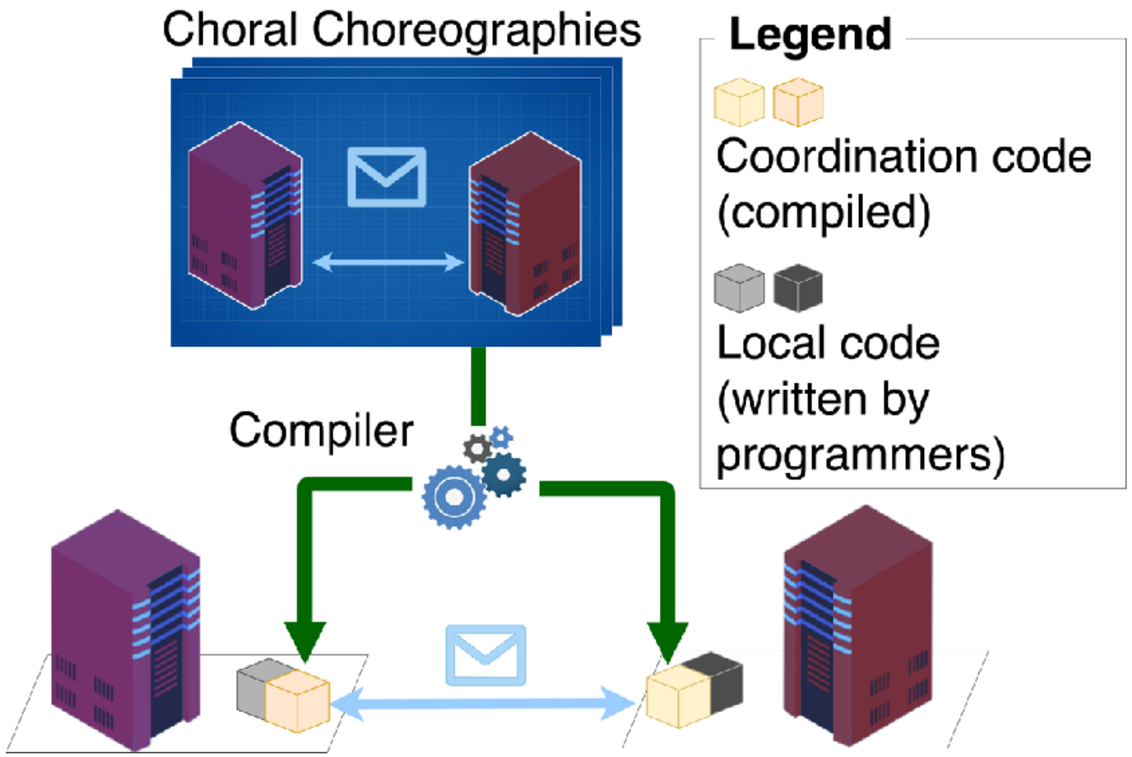
Choreographic programming

1. **User** knows $\{password\}$
2. $\mathbf{User} \rightarrow \mathbf{SSO} : \{password\};$
3. **SSO** generates $\{token\};$
4. $\mathbf{SSO} \rightarrow \mathbf{Server} : \{token\};$
3. **Server** stores $\{token\};$
5. $\mathbf{SSO} \rightarrow \mathbf{User} : \{token\}; \epsilon$



$$pwd@\mathbf{User} \rightarrow pwd@\mathbf{SSO};$$
$$token@\mathbf{SSO} = gen(pwd@\mathbf{SSO});$$
$$token@\mathbf{SSO} \rightarrow token@\mathbf{Server};$$
$$store(token@\mathbf{Server});$$
$$token@\mathbf{SSO} \rightarrow token@\mathbf{User}$$

Choreographic programming



$$pwd@\mathbf{User} \rightarrow pwd@\mathbf{SSO};$$
$$token@\mathbf{SSO} = gen(pwd@\mathbf{SSO});$$
$$token@\mathbf{SSO} \rightarrow token@\mathbf{Server};$$
$$store(token@\mathbf{Server});$$
$$token@\mathbf{SSO} \rightarrow token@\mathbf{User}$$

Choreographic programming, projection

USR

$$send(pwd, \mathbf{SSO});$$
$$token = recv(\mathbf{SSO})$$

SSO

$$pwd = recv(\mathbf{User});$$
$$token = gen(pwd);$$
$$send(token, \mathbf{Server});$$
$$send(token, \mathbf{User})$$

SRV

$$token = recv(\mathbf{SSO});$$
$$store(token)$$

$$pwd@\mathbf{User} \rightarrow pwd@\mathbf{SSO};$$
$$token@\mathbf{SSO} = gen(pwd@\mathbf{SSO});$$
$$token@\mathbf{SSO} \rightarrow token@\mathbf{Server};$$
$$store(token@\mathbf{Server});$$
$$token@\mathbf{SSO} \rightarrow token@\mathbf{User}$$

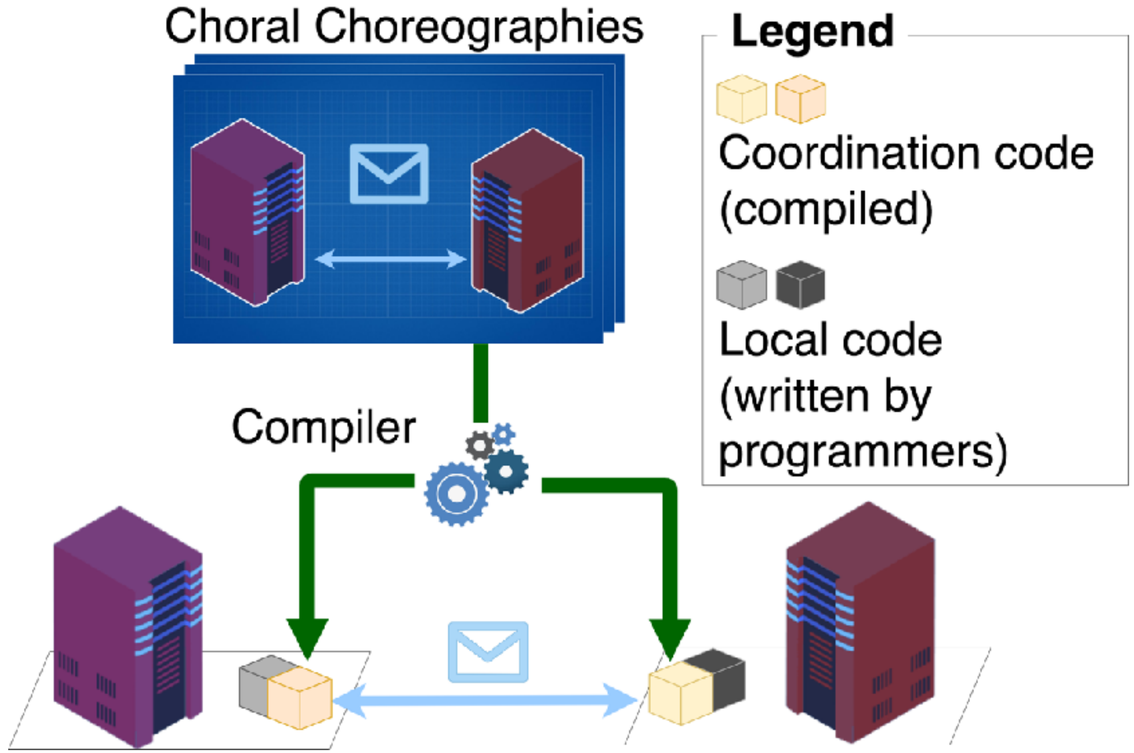
Choreographic programming, projection

USR

$$send(pwd, \mathbf{SSO});$$
$$token = recv(\mathbf{SSO})$$

SSO

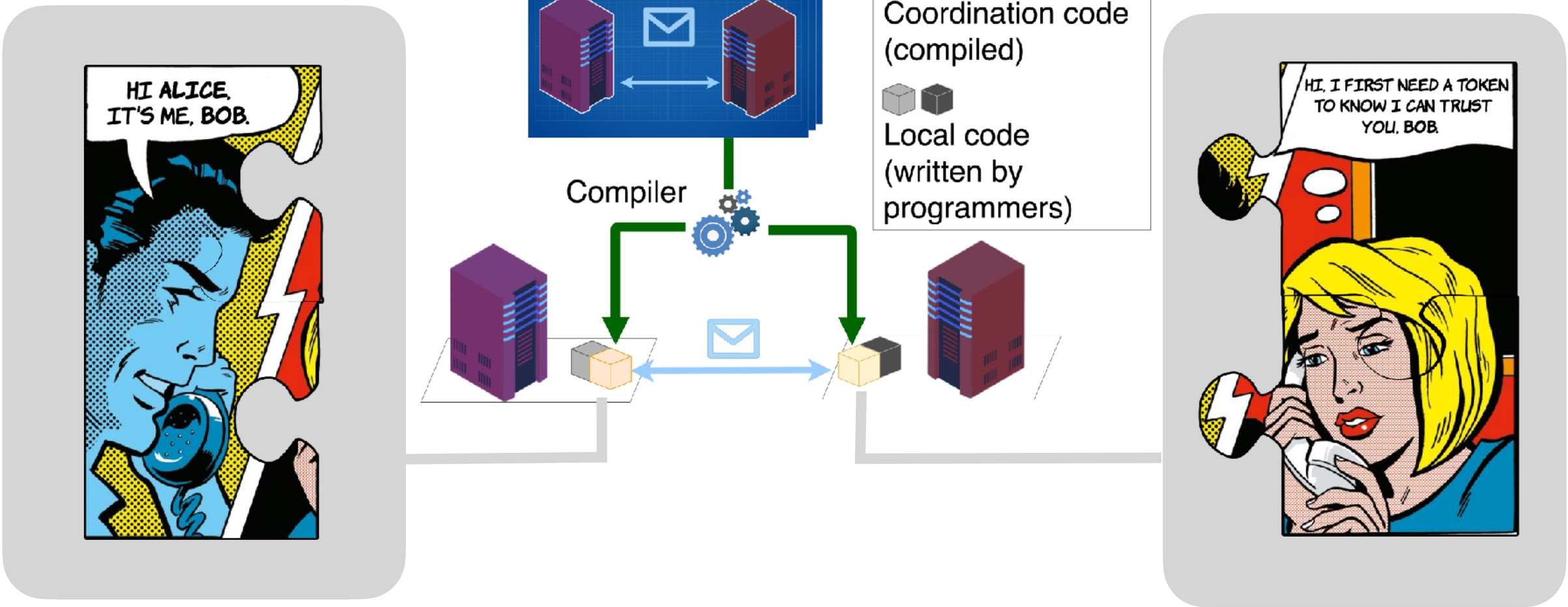
$$pwd = recv(\mathbf{User});$$
$$token = gen(pwd);$$
$$send(token, \mathbf{Server});$$
$$send(token, \mathbf{User})$$



SRV

$$token = recv(\mathbf{SSO});$$
$$store(token)$$

Choreographic programming, projection



Choreographic programming, projection

Choreographic programming, limitation

USR

$$send(pwd, \mathbf{SSO});$$
$$token = recv(\mathbf{SSO})$$

SSO

$$pwd = recv(\mathbf{User});$$
$$token = gen(pwd);$$
$$send(token, \mathbf{Server});$$
$$send(token, \mathbf{User})$$

SRV

$$token = recv(\mathbf{SSO});$$
$$store(token)$$

Modularity is limited: we generate “black box” code without an API—executable, but hardly composable in a structured way within mainstream languages.

Choreographic programming, limitation

USR

$send(pwd, \mathbf{SSO});$

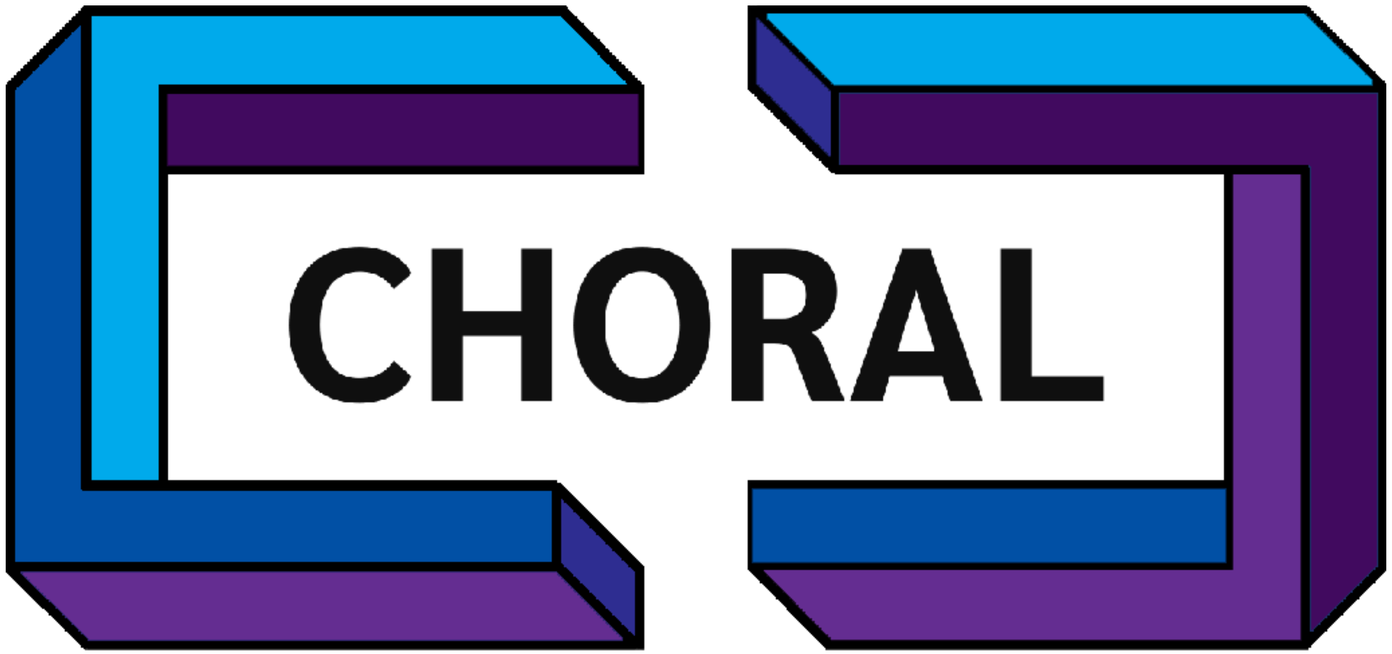
$token = recv(\mathbf{SSO})$


```
class PhotoUploader {
 void uploadPhoto( Byte[] p, String pwd ){
  Token tk = // ??? 🤔
  PhotoService.upload( tk, p );
 }
}
```

Choreographic programming, on Mainstream Tides

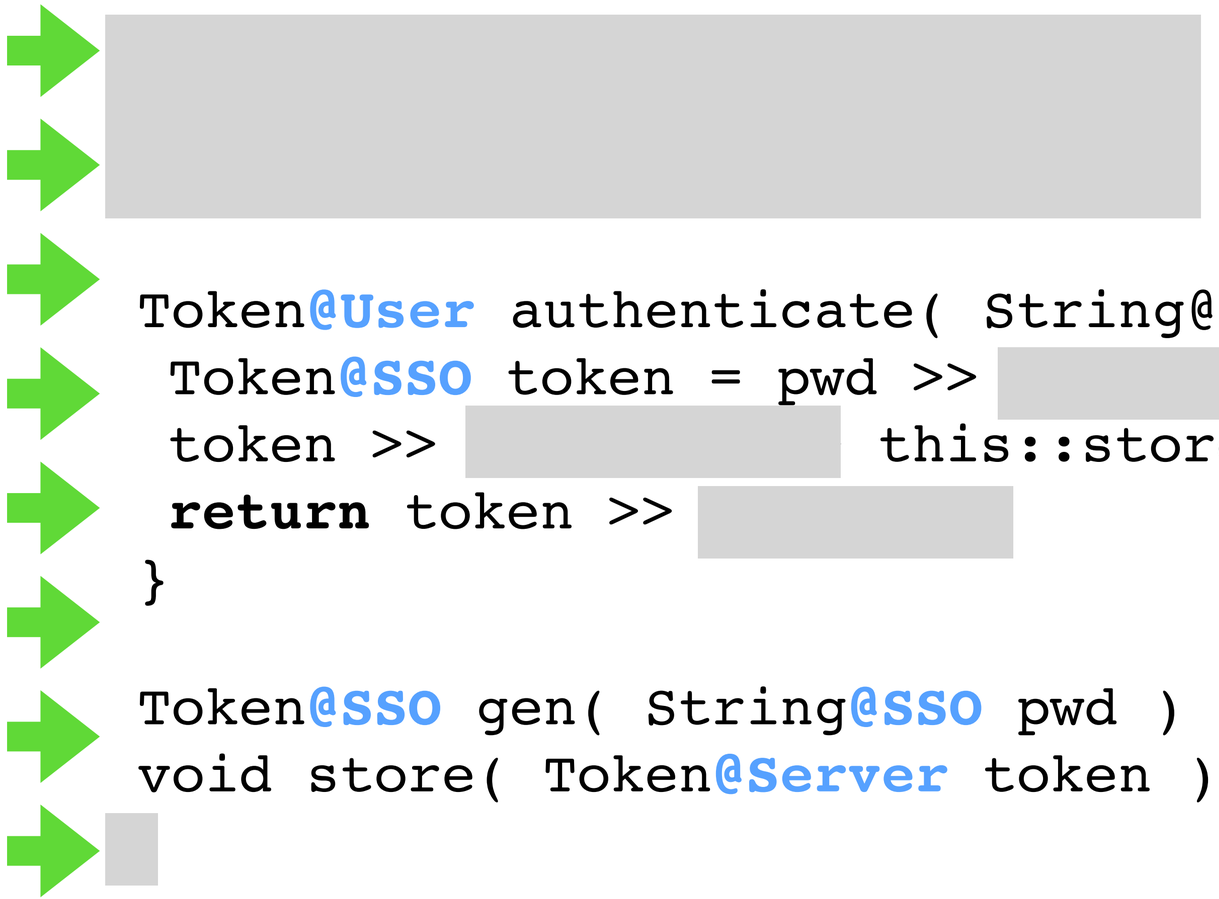
Intuition: we need to bring the salient features of the target language of the endpoints at the choreographic level.

Choreographies as Objects and Choral



Choreographic programming, on Mainstream Tides

$pwd@\mathbf{User} \rightarrow pwd@\mathbf{SSO};$
$token@\mathbf{SSO} = gen(pwd@\mathbf{SSO});$
$token@\mathbf{SSO} \rightarrow token@\mathbf{Server};$
$store(token@\mathbf{Server});$
$token@\mathbf{SSO} \rightarrow token@\mathbf{User}$



```
Token@User authenticate( String@User pwd ) {
 Token@SSO token = pwd >>                  this::gen;
 token >>                                  this::store;
 return token >>
}

Token@SSO gen( String@SSO pwd ) { . . . }
void store( Token@Server token ) { . . . }
```

Choreographic programming, on Mainstream Tides

$pwd@\mathbf{User} \rightarrow pwd@\mathbf{SSO};$
$token@\mathbf{SSO} = gen(pwd@\mathbf{SSO});$
$token@\mathbf{SSO} \rightarrow token@\mathbf{Server};$
$store(token@\mathbf{Server});$
$token@\mathbf{SSO} \rightarrow token@\mathbf{User}$

```
Channel@(User, SSO) ch1;
Channel@(SSO, Server) ch2;

Token@User authenticate( String@User pwd ) {
 Token@SSO token = pwd >> ch1::com >> this::gen;
 token >> ch2::com >> this::store;
 return token >> ch1::com;
}

Token@SSO gen( String@SSO pwd ) { . . . }
void store( Token@Server token ) { . . . }
```

Choreographic programming, on Mainstream Tides

$pwd@\mathbf{User} \rightarrow pwd@\mathbf{SSO};$
$token@\mathbf{SSO} = gen(pwd@\mathbf{SSO});$
$token@\mathbf{SSO} \rightarrow token@\mathbf{Server};$
$store(token@\mathbf{Server});$
$token@\mathbf{SSO} \rightarrow token@\mathbf{User}$

```
Channel@(User, SSO) ch1;
Channel@(SSO, Server) ch2;

Token@User authenticate( String@User pwd ) {
 Token@SSO token = gen( ch1.com( pwd ) );
 store( ch2.com( token ) );
 return ch1.com( token );
}

Token@SSO gen( String@SSO pwd ) { . . . }
void store( Token@Server token ) { . . . }
```

Choreographic programming, on Mainstream Tides

$pwd@\mathbf{User} \rightarrow pwd@\mathbf{SSO};$
$token@\mathbf{SSO} = gen(pwd@\mathbf{SSO});$
$token@\mathbf{SSO} \rightarrow token@\mathbf{Server};$
$store(token@\mathbf{Server});$
$token@\mathbf{SSO} \rightarrow token@\mathbf{User}$

```
class Auth@(User, Server, SSO) {
 Channel@(User, SSO) ch1;
 Channel@(SSO, Server) ch2;

 Token@User authenticate( String@User pwd ) {
  Token@SSO token = gen( ch1.com( pwd ) );
  store( ch2.com( token ) );
  return ch1.com( token );
 }

 Token@SSO gen( String@SSO pwd ) { . . . }
 void store( Token@Server token ) { . . . }
}
```

Choreographic programming, on Mainstream Tides

Projection to Java code/libraries

USR

```
class Auth_User {
 Channel_A ch1;

 Token authenticate( String pwd ) {
  gen( ch1.com( pwd ) );
  store( Unit.id );
  return ch1.com( Unit.id );
 }
 Unit gen( Unit pwd ) { … }
 void store( Unit token ) { … }
}
```

Choreographic programming, on Mainstream Tides

Projection to Java code/libraries

USR

```
class Auth_User {
 Channel_A ch1;

 Token authenticate( String pwd ) {
  gen( ch1.com( pwd ) );
  store( Unit.id );
  return ch1.com( Unit.id );
 }
 Unit gen( Unit pwd ) { … }
 void store( Unit token ) { … }
}
```

LOCAL CODE

```
class PhotoUploader {
 void uploadPhoto( Byte[] p, String pwd ){
  Channel_A ch = SSO.connect();
  Auth_User au = new Auth_User( ch );
  Token tk = au.authenticate( pwd );
  PhotoService_Client.upload( tk, p );
 }
}
```

Choreographic programming, on Mainstream Tides

Projection to Java code/libraries

USR

```
class Auth_User {
 Channel_A ch1;

 Token authenticate( String pwd ) {
  gen( ch1.com( pwd ) );
  store( Unit.id );
  return ch1.com( Unit.id );
 }
 Unit gen( Unit pwd ) { … }
 void store( Unit token ) { … }
}
```

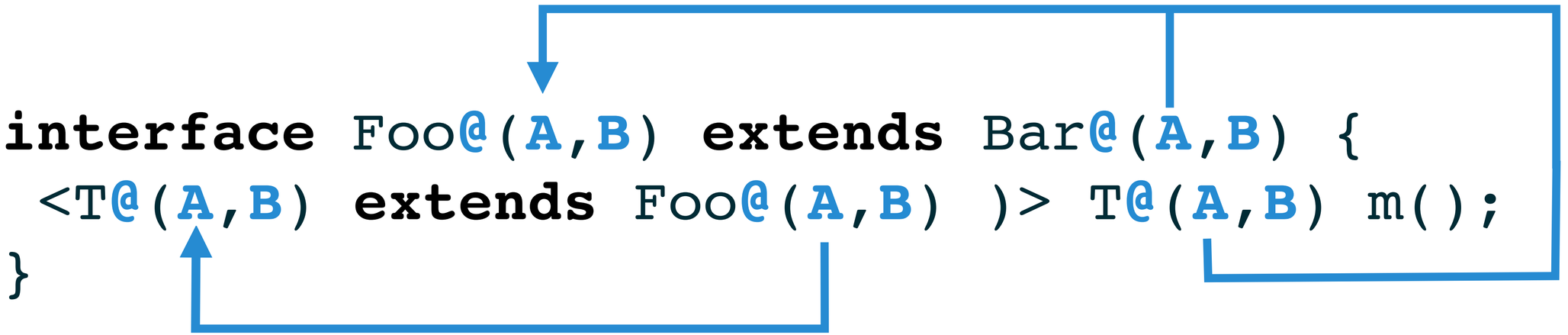
SSO

```
class Auth_SSO {
 Channel_B ch1;
 Channel_A ch2;

 Unit authenticate( Unit pwd ) {
   Token token = gen( ch1.com( Unit.id ) );
   store( ch2.com( token ) );
   return ch1.com( token );
 }
 Token gen( String pwd ) { … }
 void store( Unit token ) { … }
}
```

SRV

```
class Auth_Server {
 Channel_B ch2;

 Unit authenticate( Unit pwd ) {
  gen( Unit.id );
  store( ch2.com( Unit.id ) );
  return Unit.id;
 }
 Unit gen( Unit pwd ) { … }
 void store( Token token ) { … }
}
```

Choral: inspiration and intuition, type parameters



Choral, Channels

```
interface DiDataChannel@(A, B)<T@X> {
 <S@Y extends T@Y> S@B com(S@A m);
}
```

Choral, Channels

```
interface DiDataChannel@(A, B)<T@X> {
 <S@Y extends T@Y> S@B com(S@A m);
}

interface BiDataChannel@(A, B)<T@X, R@Y> extends
   DiDataChannel@(A, B)<T>, DiDataChannel@(B, A)<R>
{ }

interface SymDataChannel@(A, B)<T@X> extends
   BiDataChannel@(A, B)<T,T>
{ }
```

Choral, knowledge of choice

DOES NOT COMPILE

```
DiDataChannel@(A, B)<Item@X> ch;
Iterator@A<Item> it;
Consumer@B<Item> consumer;
consumeItems() {
  if ( it.hasNext() ){
    it.next() >> ch::<Item>com >> consumer::accept;
    consumeItems();
  }
}
```

We need to let **B** know the outcome of the choice taken by **A**

Choral, knowledge of choice

We define a method-level annotation `@SelectionMethod`, which developers can apply only to methods that can transmit instances of enumerated types between roles (the compiler checks for this condition). Our compiler assumes that implementations of such annotated methods return at the receiver the same value given at the sender.

We can thus define a family of channels able to transmit choices

```
interface DiSelectChannel@(A, B){
  @SelectionMethod
  <T@X extends Enum@X<T@X>> T@B select(T@A m);
}

interface DiChannel@(A, B)<T@X> extends
   DiDataChannel@(A, B)<T>, DiSelectChannel@(A,B){
}
```

Choral, knowledge of choice

COMPILES

```
enum Choice@Y{ GO, STOP }
DiChannel@(A, B)<Item> ch;
Iterator@A<Item> it;
Consumer@B<Item> consumer;
consumeItems() {
  if ( it.hasNext() ){
    Choice@A.GO >> ch::select;
    it.next() >> ch::<Item>com >> consumer::accept;
    consumeItems();
  } else {
    Choice@A.STOP >> ch::select;
}
```

Choral, syntax

Literals	*lit*	::=	`null`@(A) | `true`@A | `false`@A | `"a"`@A | ... | `1`@A | ...
Program	*P*	::=	*P* *Interface* | *P* *Class* | *P* *Enum* | *P* *EOF*
Enum	*Enum*	::=	*AN* $\overline{MD}$ `enum` *id*@A{ $\overline{id}$ }
Interface	*Interface*	::=	*AN* $\overline{MD}$ `interface` *id*@(A)⟨$\overline{FTP}$⟩ `extends` *TE*, $\overline{TE}$ { $\overline{MDef}$; }
Annotation	*AN*	::=	@*id*($\overline{id = lit}$)
Modifiers	*MD*	::=	`public` | `protected` | `private` | `abstract` | `final` | `static`
Formal Type Param.	*FTP*	::=	*id*@(A) `extends` *TE* `&` $\overline{TE}$
Type Expr.	*TE*	::=	*id*⟨$\overline{TE}$⟩ | *id*@(A)⟨$\overline{TE}$⟩ | `void`
Method Def.	*MDef*	::=	*AN* $\overline{MD}$ ⟨$\overline{FTP}$⟩ *TE* *id* ($\overline{TE\ id}$)
Class	*Class*	::=	*AN* $\overline{MD}$ `class` *id*@(A)⟨$\overline{FTP}$⟩ `extends` *TE* `implements` *TE*, $\overline{TE}$ { $\overline{CField}$ $\overline{CConst}$ $\overline{MDef;}$ $\overline{MDef\{Stm\}}$ }
Class Field.	*CField*	::=	*AN* $\overline{MD}$ *TE* $\overline{id}$;
Class Con.	*CConst*	::=	*AN* $\overline{MD}$ ⟨$\overline{FTP}$⟩ *id* ($\overline{TE\ id}$) { *Stm* }

Choral, syntax

Statement	*Stm*	::=	*nil* | `return` *Exp* `;` | *Exp* `;` *Stm* | *TE* *id* `=` *Exp* `;` *Stm*
			| *Exp* *AsgOp* *Exp* `;` *Stm* | `if` `(` *Exp* `)` `{` *Stm* `}` `else` `{` *Stm* `}` *Stm*
			| `{` *Stm* `}` *Stm* | `try` `{` *Stm* `}` `catch` `(` *TE* *id* `)` `{` *Stm* `}` *Stm*
			| `switch` `(` *Exp* `)` `{` `case` *SwArg* `->` `{` *Stm* `}` `default` `->` `{` *Stm* `}` `}` *Stm*
Switch Arg.	*SwArg*	::=	*id* | *lit*
Expression	*Exp*	::=	*lit* | *FAcc* | *Exp* *BinOp* *Exp* | *Exp* `.` *Exp* | $\langle \overline{TE} \rangle$ *id* $(\overline{Exp})$
			| `new` $\langle \overline{TE} \rangle$ *id* `@` `(` $\overline{A}$ `)` $\langle \overline{TE} \rangle$ $(\overline{Exp})$ | *id* `@` `(` $\overline{A}$ `)` `.` $\langle \overline{TE} \rangle$ *id* $(\overline{Exp})$ | *Exp* `>>` $\overline{EChain}$
Field Acc.	*FAcc*	::=	*id* | *id* `@` `(` $\overline{A}$ `)` `.` *id*
Exp. Chain	*EChain*	::=	*FAcc* $\overline{\texttt{.}\ id}$ `::` *id* | *id* `@` `(` $\overline{A}$ `)` $\langle \overline{TE} \rangle$ `::` `new`
Assign Op.	*AsgOp*	∈	{`=`, `+=`, `-=`, `*=`, `/=`, `&!=`, `|!=`, `%!=`}
Binary Op.	*BinOp*	∈	{`||`, `&&`, `|`, `&`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `%`}

Choral: types, highlights

Compiler error

```
Integer@A x = "foo"@A; // matching role, apply the same rules as Java
------------^
Incompatible types: expecting 'Integer@A' found 'String@A'.
```

Choral: types, highlights

Compiler error

```
void m (SymChannel@(A,B)<T> x) {
  DiChannel@(A,B)<T> y = x;  // ok, SymChannel@(A,B)<T> extends DiChannel@(B,A)<T>
  SymChannel@(B,A)<T> z = x; // not ok, mismatching roles
-----------------------^
Incompatible types: expecting 'SymChannel@(B,A)<T>' found 'SymChannel@(A,B)<T>'.
```

Choral: types, highlights

Compiler error

```
interface SymChannel@(A,B)<T@X> extends SymChannel@(B,A)<T> { /* ... */ }
-----------------------------------^
Cyclic inheritance: 'SymChannel' cannot extend 'SymChannel'.
```

Since in the compiled Java we would have

```
interface SymChannel_A<T> extends SymChannel_B<T> { /* ... */ } // Projection for A
interface SymChannel_B<T> extends SymChannel_A<T> { /* ... */ } // Projection for B
```

Choral: types, highlights

Compiler error

```
abstract void m(DiChannel@(A,A)<String> channel);
----------------------------^
Illegal type instantiation: role 'A' must play exactly one role in 'DiChannel'.
```

Choral: types, highlights

Compiler error

```
class Foo@(A,B) {
  void m(Char@B x) { /* ... */ }  // ok:     void m() at A and void m(Char x) at B
  void m(Char@A x) { /* ... */ }  // ok:     void m(Char x) at A and void m() at B
  void m(Long@A x) { /* ... */ }  // not ok: void m(Long x) at A and void m() at B
-------^
Illegal overload: 'm(Long@A x)' and 'm(Char@A x)' have the same signature for role 'B'.
```

ChoralUnit: distributed unit testing

```
class CheckTokenValidity@(U,S,SSO){
 @Test
 public static void test(){
  Auth@(U,S,SSO) a = new Auth@(U,S,SSO)(
   TestUtils@(U,SSO).newLocalChannel(),
   TestUtils@(SSO,S).newLocalChannel()
  );
  SymChannel@(U,S) ch = TestUtils@(U,S).newLocalChannel();
  Assert@U.assertTrue( “Invalid Token”@U,
   “myPassword”@U >> a::authenticate >> ch::com
   >> a::isValidToken >> ch::com );
 }
}
```

Choral: evaluation (1 of 4)

The Karatsuba Algorithm

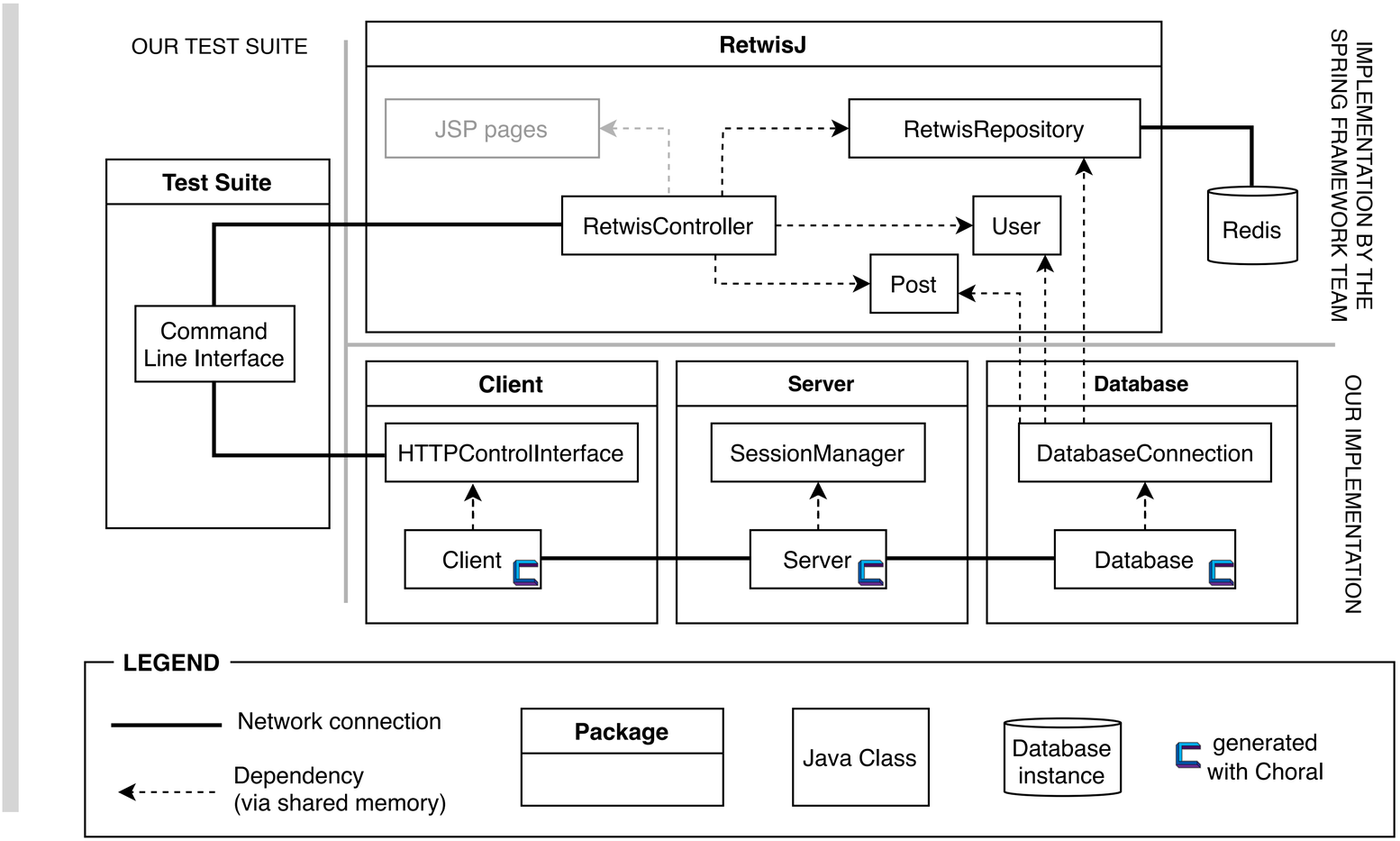
```
public static Long multiply(Long n1, Long n2) {
  if (n1 < 10 || n2 < 10) {
    return n1 * n2;
  } else {
    Double m = Math.max(Math.log10(n1), Math.log10(n2)) + 1;
    Integer m2 = Double.valueOf(m / 2).intValue();
    Integer splitter = Double.valueOf(Math.pow(10, m2)).intValue();
    Long h1 = n1 / splitter; Long l1 = n1 % splitter;
    Long h2 = n2 / splitter; Long l2 = n2 % splitter;
    Long z0 = Karatsuba.multiply(l1, l2);
    Long z2 = Karatsuba.multiply(h1, h2);
    Long z1 = Karatsuba.multiply(l1 + h1, l2 + h2) - z2 - z0;
    return z2 * splitter * splitter + z1 * splitter + z0;
  }
}
```

```
public static Long@A multiply (Long@A n1, Long@A n2,
  SymChannel@(A, B)<Long> ch_AB,
  SymChannel@(B, C)<Long> ch_BC,
  SymChannel@(C, A)<Long> ch_CA) {
  if (n1 < 10@A || n2 < 10@A) {
    ch_AB.<Choice>select(Choice@A.DONE); ch_CA.<Choice>select(Choice@A.DONE);
    return n1 * n2;
  } else {
    ch_AB.<Choice>select(Choice@A.REC); ch_CA.<Choice>select(Choice@A.REC);
    Double@A m = Math@A.max(Math@A.log10(n1), Math@A.log10(n2)) + 1@A;
    Integer@A m2 = Double@A.valueOf(m / 2@A).intValue();
    Integer@A splitter = Double@A.valueOf(Math@A.pow(10@A, m2)).intValue();
    Long@A h1 = n1 / splitter; Long@A l1 = n1 % splitter;
    Long@A h2 = n2 / splitter; Long@A l2 = n2 % splitter;
    Long@A z0 = Karatsuba@(B, C, A)
      .multiply(ch_AB.<Long>com(l1), ch_AB.<Long>com(l2), ch_BC, ch_CA, ch_AB)
      >> ch_AB::<Long>com;
    Long@A z2 = Karatsuba@(C, A, B)
      .multiply(ch_CA.<Long>com(h1), ch_CA.<Long>com(h2), ch_CA, ch_AB, ch_BC)
      >> ch_CA::<Long>com;
    Long@A z1 = Karatsuba@(A, B, C)
      .multiply(l1 + h1, l2 + h2, ch_AB, ch_BC, ch_CA) - z2 - z0;
    return z2 * splitter * splitter + z1 * splitter + z0;
  }
}
```

Choral: evaluation (2 of 4)

Comparison with Akka and usage in the real world

Program (LOC)	DistAuth	MergeSort	Karatsuba
Choral	47	62	26
Java/ Choral-generated	115	231	85
Java + Akka	234	166	118



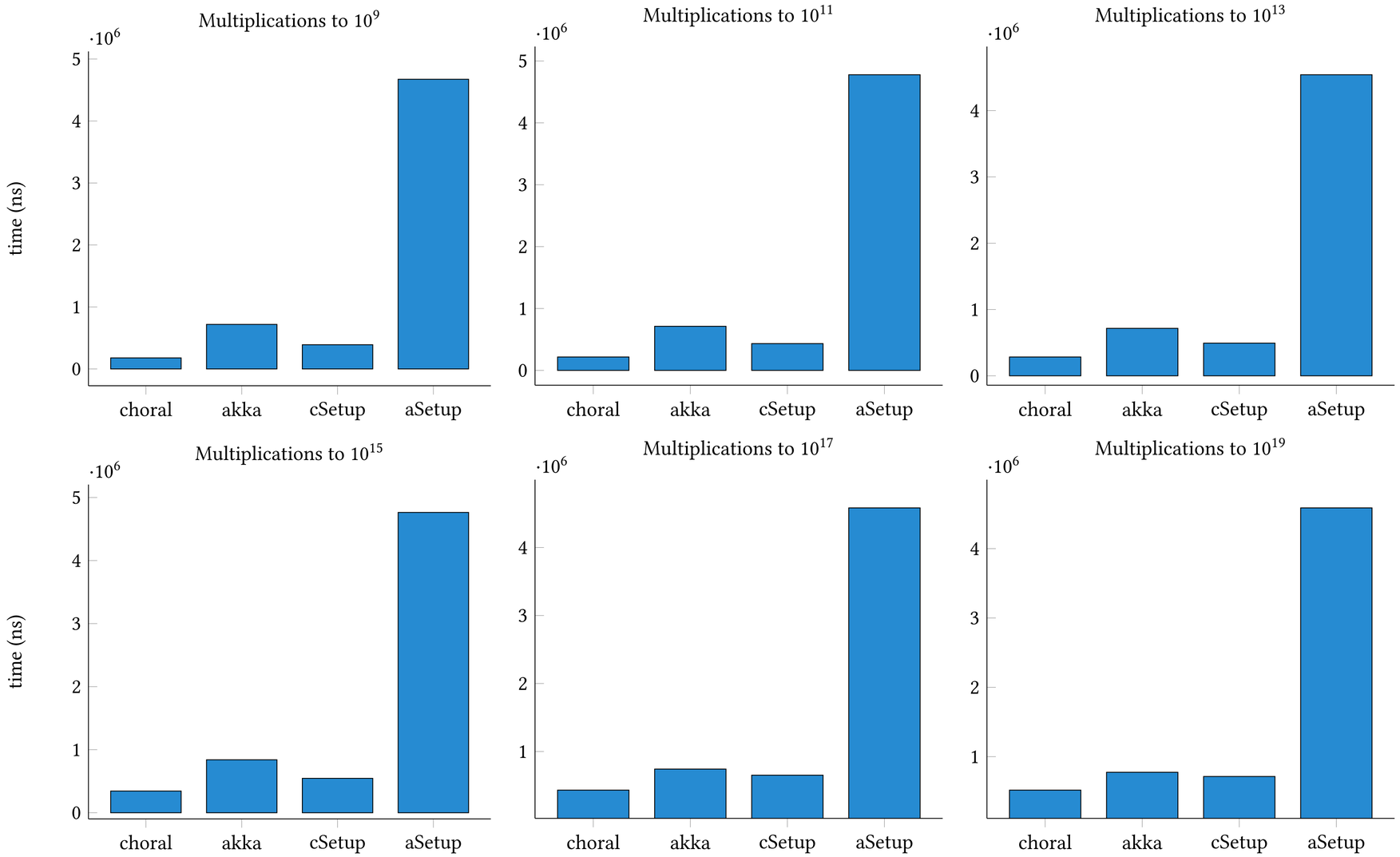
Choral: evaluation (3 of 4)

Performance of the compiler

Program	Choral (LOC)	# Roles	# Conditionals	Java (LOC)	Size Increase (%)	Type Checking (ms)	Projection (ms)
HelloRoles	8	2	0	12	50%	3.409	0.161
ConsumeItems	11	2	1	35	218%	3.332	0.287
BuyerSellerShipper	35	3	2	126	260%	5.047	0.839
DistAuth	47	3	1	115	145%	7.496	0.680
VitalsStreaming	38	2	1	71	87%	4.574	0.382
DiffieHellman	10	2	0	30	300%	4.614	0.475
MergeSort	62	3	4	231	273%	5.832	3.845
QuickSort	63	3	3	199	216%	5.516	3.080
Karatsuba	26	3	1	85	227%	4.857	1.874
DistAuth5	57	5	1	197	246%	7.522	1.175
DistAuth10	82	10	1	402	390%	7.964	2.956

Choral: evaluation (4 of 4)

Runtime performance comparison of the Karatsuba algorithm



Future work (on Choral)

- Choral supports classes, interfaces, generics, inheritance, and method overloading. Java Lambdas can help with language ergonomics.
- `class Auth1@(U,S) extends Auth@(U,S,SSO)` and `class Auth1@(U,S,SSO,Logger) extends Auth@(U,S,SSO)`
-
```
class LoadBalancer@(Prod,W1,W2){
  [W in (W1,W2) Task@W] dist(Task@Prod t){} }
```
-
```
class StarChannel@(A,*Bs){
[B for Bs String@B] scatter(String@A m){}
String@A gather([B for Bs String@B] m){} }
```
- Choral offer some basic support for error handling (`try-catch` on subtypes of `Exception@A`, unchecked). What is the handling semantics of `Exception@(A,B,C)`?
- Can we compile to multiple (polyglot) target languages?
- Can we dispense `@selectMethod`s when other `com`s already exist?

Future work (in general)

- **Featherweight Choral**
  - A formal model for Choral, we started this with mini-Choral in ECOOP 2021. Other references: choreographic λ-calculus, Coq formalisation

- Exploring the **foundations of choreographies**:
  - axiomatisations and denotational semantics for choreographies
  - categorical models of choreographies, morphism of projection (and extraction)

- **Probabilistic choreographies**, e.g., to estimate confidence intervals in protocols with failures