Designing, Verifying and Monitoring Protocols

inspired by Scribble

Versions presented:
Luxembourg-Singapore Workshop on Security and Privacy — 2 March 2016
InfoLab21, Lancaster University, United Kingdom — 17 November 2015
Queen Mary University of London, United Kingdom — 11 November 2015
PSI: 10th Ershov Informatics Conference, Kazan, Russia — 25–27 August 2015

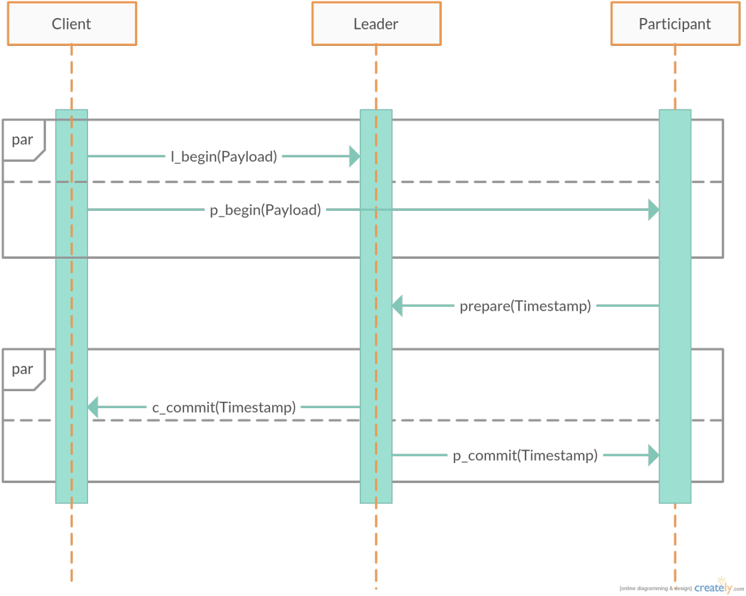
Ross Horne

School of Computer Engineering, Nanyang Technological University, Singapore

2 March 2016

Sessions in Distributed Systems

Client driven two phase commit (2PC) as a *sequence diagram*.



Sessions in Distributed Systems

Client driven two phase commit (2PC) as a *global session type* (based on Scribble[1]).

```
{  par
      p_begin(Payload) from Client to Participant
   and
      l_begin(Payload) from Client to Leader
} ;

prepare(Timestamp) from Participant to Leader ;

{  par
      c_commit(Timestamp) from Leader to Client
   and
      p_commit(Timestamp) from Leader to Participant
}
```

---

[1] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. *Scribbling interactions with a formal foundation.* In Distributed Computing and Internet Technology, pages 5575. Springer, 2011.

Sessions in Distributed Systems

Local session types for roles **Client**, **Participant** and **Leader** (based on Scribble).

Client:

```
{ par ~p_begin(Payload) to Participant
  and ~l_begin(Payload) to Leader
} ;
c_commit(Timestamp) from Leader
```

Leader:

```
l_begin(Payload) from Client ;
prepare(Timestamp) from Participant ;
{ par ~p_commit(Timestamp) to Participant
  and ~c_commit(Timestamp) to Client
}
```

Participant:

```
p_begin(Payload) from Client ;
~prepare(Timestamp) to Leader ;
p_commit(Timestamp) from Leader
```

A Semantics for Multi-party Session Types

- How do we know that the projection is correct?
- How do we know when a protocol of one type can do everything that a protocol of another type can do?
- How can we determine when a collection of local types are compatible?

We need a **semantics**!

Sessions in Distributed Systems

Local session types for roles **Client**, **Participant** and **Leader**.

Client:

```
{  par ∼p_begin(Payload) to Participant
   and ∼l_begin(Payload) to Leader
} ;
c_commit(Timestamp) from Leader
```

Leader:

```
l_begin(Payload) from Client ;
prepare(Timestamp) from Participant ;
{  par ∼p_commit(Timestamp) to Participant
   and ∼c_commit(Timestamp) to Client
}
```

Participant:

```
p_begin(Payload) from Client ;
∼prepare(Timestamp) to Leader ;
p_commit(Timestamp) from Leader
```

Multi-party Compatibility

```
par
   {  par ~p_begin(Payload) to Participant
      and ~l_begin(Payload) to Leader
   };
   c_commit(Timestamp) from Leader
and
   l_begin(Payload) from Client ;
   prepare(Timestamp) from Participant ;
   {  par ~p_commit(Timestamp) to Participant
      and ~c_commit(Timestamp) to Client
   }
and
   p_begin(Payload) from Client ;
   ~prepare(Timestamp) to Leader ;
   p_commit(Timestamp) from Leader
```

Multi-party Compatibility

```
{   par ~p_begin(Payload) to Participant
    and p_begin(Payload) from Client

    and ~l_begin(Payload) to Leader
    and l_begin(Payload) from Client
} ;
{   par ~prepare(Timestamp) to Leader
    and prepare(Timestamp) from Participant
} ;
{   par ~p_commit(Timestamp) to Participant
    and p_commit(Timestamp) from Leader

    and ~c_commit(Timestamp) to Client
    and c_commit(Timestamp) from Leader
}
```

Multi-party Compatibility

$\{ \}$

A Semantics for Session Types in the Calculus of Structures

atomic interaction
$\mathtt{par}\ {\sim}A$ and $B \longrightarrow \{\}$ only if A is a subsort of B

seq
$\mathtt{par}\ \{\ T\ ;\ U\ \}\ \mathtt{and}\ \{\ V\ ;\ W\ \} \longrightarrow \{\ \mathtt{par}\ T\ \mathtt{and}\ V\ \}\ ;\ \{\ \mathtt{par}\ U\ \mathtt{and}\ W\ \}$

switch
$\mathtt{par}\ \{\ \mathtt{sync}\ T\ \mathtt{and}\ U\ \}\ \mathtt{and}\ V \longrightarrow \mathtt{sync}\ T\ \mathtt{and}\ \{\ \mathtt{par}\ U\ \mathtt{and}\ V\ \}$

left choice
$T\ \mathtt{or}\ U \longrightarrow T$

right choice
$T\ \mathtt{or}\ U \longrightarrow U$

tidy
$\{\}\ \&\ \{\} \longrightarrow \{\}$

external choice
$\mathtt{par}\ T\ \mathtt{and}\ \{\ U\ \&\ V\ \} \longrightarrow \{\ \mathtt{par}\ T\ \mathtt{and}\ U\ \}\ \&\ \{\ \mathtt{par}\ T\ \mathtt{and}\ V\ \}$

medial
$\{\ T\ ;\ U\ \}\ \&\ \{\ V\ ;\ W\ \} \longrightarrow \{\ T\ \&\ V\ \}\ ;\ \{\ U\ \&\ W\}$

---

context closure
$\mathcal{C}\{\ T\ \} \longrightarrow \mathcal{C}\{\ U\ \}$ only if $T \longrightarrow U$

congruence
$T \longrightarrow U$ only if $T \equiv U$

$(\mathcal{T}, ;, \{\})$ is a monoid and $(\mathcal{T}, par, \{\})$ and $(\mathcal{T}, sync, \{\})$ are commutative monoids.

Proof and Multi-party Compatibility

Definition (Proof)

A sequence of rewrites that ends with the unit ($\{\ \}$) is a **proof**. [2]

Definition (Multi-party compatibility)

If the parallel composition of all roles (and channels) is provable then the local protocols are **multi-party compatible**. [3] [4]

Proposition

The multiset of projections from any global protocol to it's local protocols for roles (and channels) is multi-party compatible.

---

[2] Alessio Guglielmi. *A system of interaction and structure*. ACM ToCL, 8, 2007.

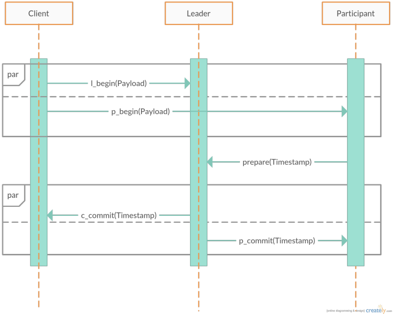
[3] Kohei Honda. *Types for dyadic interaction*. In CONCUR93, pages 509-523, 1993.

[4] Kohei Honda, Nobuko Yoshida, and Marco Carbone. *Multiparty asynchronous session types.* ACM SIGPLAN Notices, 43(1):273284, 2008.

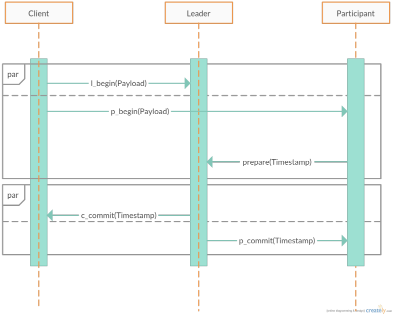
Subtyping

Which protocol is a subtype of the other protocol?

2PC :



2PC$'$:



I.e., can one protocol do everything that another protocol can do in every context?

Check for Subtyping

Definition

A local type T is a subtype of local type U, written $T \leq U$, if and only if `par` $\sim T$ `and` U is provable.

Firstly apply *De Morgan properties* to find the complement of **Leader**.

```
Leader :    l_begin(Payload) from Client ;
            prepare(Timestamp) from Participant ;
            {
                par ~p_commit(Timestamp) to Participant
                and ~c_commit(Timestamp) to Client
            }

~Leader :   ~l_begin(Payload) from Client ;
            ~prepare(Timestamp) from Participant ;
            {
                sync p_commit(Timestamp) to Participant
                and c_commit(Timestamp) to Client
            }
```

Check for Subtyping

Definition

A local type T is a subtype of local type U, written $T \leq U$, if and only if par $\sim T$ and U is provable.

```
par   ~l_begin(Payload) from Client ;
      ~prepare(Timestamp) from Participant ;
      {
          sync p_commit(Timestamp) to Participant
          and c_commit(Timestamp) to Client
      }

and   {
          par prepare(Timestamp) from Participant
          and l_begin(Payload) from Client
      } ;
      {
          par ~p_commit(Timestamp) to Participant
          and ~c_commit(Timestamp) to Client
      }
```

The above is provable, hence $\mathbf{Leader} \leq \mathbf{Leader}'$. Hence $\mathbf{Leader}'$ can do everything **Leader** can do in any context.

For global protocols apply subtyping point-wise, hence $\mathbf{2PC} \leq \mathbf{2PC}'$.

Example 2PC with the option for the participant to abort.

Client″ :

```
{   par ~p_begin(Payload) to Participant
    and ~l_begin(Payload) to Leader
} ;
{   commit(Timestamp) from Leader
    or
    c_abort(Error) from Leader
}
```

Participant″ :

```
p_begin(Payload) from Client ;
{   {   ~prepare(Timestamp) to Leader ;
        p_commit(Timestamp) from Leader
    }
    &
    ~p_abort(Error) to Leader
}
```

Leader″ :

```
l_begin(Payload) from Client;
{   {   prepare(Timestamp) from Participant ;
        {   par ~p_commit(Timestamp) to Participant
            and ~c_commit(Timestamp) to Client
        }
    } or {
      p_abort(Error) from Participant ;
      ~c_abort(Error) to Client
    }
}
```

Due to internal choice **Leader** $\leq$ **Leader″** and **Client** $\leq$ **Client″**.
However, due to external choice **Participant″** $\leq$ **Participant**.

Coherence

Definition (Coherence)

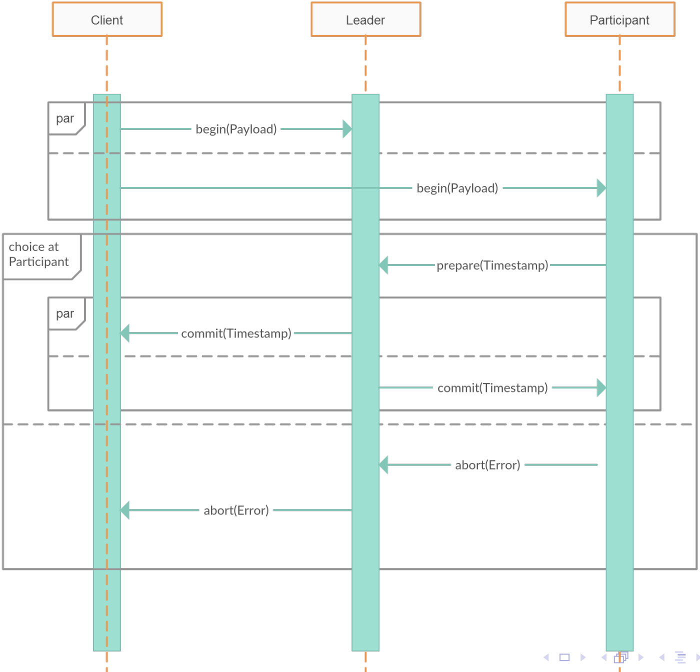
A multiset of local types $(T_i)_{i \in I}$, where I is a set of roles and channels, is *coherent* (with respect to G) if there exists a global type G such that for all $i \in I$, $G \upharpoonright_i \leq T_i$.

Leader$''$, *Participant*$''$ and *Client*$''$ (plus channels) are coherent with respect to **2PC**$''$:

```
par p_begin(Payload) from Client to Participant
and l_begin(Payload) from Client to Leader ;

choice at Participant {
          prepare(Timestamp) from Participant to Leader ;
          par c_commit(Timestamp) from Leader to Client
          and p_commit(Timestamp) from Leader to Participant
 } or {
          p_abort(Error) from Participant to Leader ;
          c_abort(Error) from Leader to Client
 }
```

Coherence


Client
Leader
Participant
par
begin(Payload)
begin(Payload)
choice at Participant
prepare(Timestamp)
par
commit(Timestamp)
commit(Timestamp)
abort(Error)
abort(Error)

Interoperability: the Sync Operator

- The Digital Ocean API can create instances in separate zones using one messages.
- The Google Compute Engine API requires a separate message for each zone.

The protocol below is part of a mediator between the APIs of the two Cloud providers.

```
sync post(JSON) from Client
and  {    par post1(JSON) to Server
          and post2(JSON) to Server } ;

{    {    sync alert(Error) from Server
          and anything
          and alert(Error) to Client }
     or
     {    sync response1(JSON) from Server
          and response2(JSON) from Server
          and response(JSON) to Client }
}
```

The sync operator is used to synchronise inputs from the servers.

Interoperability: the Sync Operator

How do I know the mediator protocol is correct?

```
Digital Ocean Client :    ~post(JSON) to Server ;
                          {    response(JSON) from Mediator
                               or
                               alert(Error) from Mediator
                          }

Mediator :     sync post(JSON) from Client
               and  {    par ~post1(JSON) to Server
                         and ~post2(JSON) to Server } ;

               {    {    sync alert(Error) from Server
                         and anything
                         and ~alert(Error) to Client }
                    or
                    {    sync response1(JSON) from Server
                         and response2(JSON) from Server
                         and ~response(JSON) to Client }
               }

2 × Google Compute Server :    post(JSON) from Mediator ;
                               {    ~response(JSON) to Mediator
                                    &
                                    ~alert(Error) to Mediator
                               }
```

Subsorting

The subtyping relation agrees with standard subtyping for I/O types. Assume the following subsort relation holds:

$$nat \leq int$$

The following hold:

- $\sim c(int)$ `to` $P \leq \sim c(nat)$ `to` P (contravarience).
  We can send something more specific (nat) when something more general (int) is expected.
- $c(nat)$ `from` $P \leq c(int)$ `from` P (covarience)
  We can be ready to receive something more general (int), when something more specific (nat) arrives.

Any preorder, e.g. subtyping for XML Schema, can be used for subsorting.

Properties of Subtyping: Cut Elimination

Theorem (Cut Elimination)

If $\mathcal{C}\{\ \mathit{sync}\ T\ \mathit{and}\ {\sim}T\ \}$ *is provable, then* $\mathcal{C}\{\ \{\ \}\ \}$ *is provable.*

[snip: 70 pages of proof] [5]

Corollary (Transitivity)

Subtyping is transitive, i.e. if $T \leq U$ *and* $U \leq V$*, then* $T \leq V$*.*

Corollary

Any coherent multiset of local types, is multiparty compatible.

Theorem (Feasibility)

Deciding the provability of a local type is a PSPACE-complete problem.

---

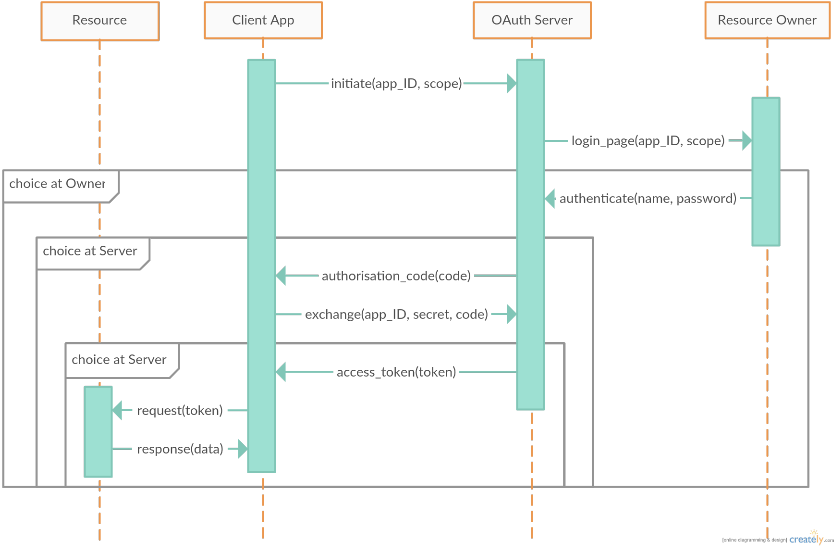
[5] Ross Horne. *The consistency and complexity of multiplicative additive system virtual*. Scientific Annals of Computer Science, 25(2):245-316, 2015.

Applications to Security and Future Collaboration

- Monitoring: Runtime monitors generated from local session types can be used to detect when a participant violates permitted protocols. Scenarios include:
    - distributed systems spanning organisation boundaries, such as a distributed database with replicas in multiple Cloud providers.
    - virtualization, where virtual machines are leased for a particular purpose only.
    - microvirtualization, where untrusted software is executed safely in an isolated process.
- Type checking: Security protocols themselves can be specified using session types. For example, an implementation of a client in an OAuth protocol can be checked against the local type for clients to ensure conformance.
- Verification: Dependently typed extensions are sufficiently powerful to be used to prove the correctness of security protocols themselves. Attacks can be discovered and the absence of certain attacks can be certified.

Example of Session Types for OAuth: Globally

OAuth protocol as a *sequence diagram*.



Example of Session Types for OAuth: Locally

App :

```
~initiate(app_ID, scope) to Server ;
{} or {
 authorisation_code(code) from Server ;
 ~exchange(app_ID, secret, code) to Server ;
 {} or {
  access_token(token) from Server ;
  ~request(token) to Resource ;
  response(data) from Resource
 }
}
```

Server :

```
initiate(app_ID, scope) from App ;
~login_page(app_ID, scope) to Owner ;
{} or {
 authenticate(name, password) from Owner ;
 {} & {
  ~authorisation_code(code) to App ;
  exchange(app_ID, secret, code) from App ;
  {} & {
   ~access_token(token) to App
  }
 }
}
```

Resource :

```
{} or {
 request(token) from App ;
 ~response(data) to App
}
```

Owner :

```
login_page(app_ID, scope) from Server ;
{} & {
 ~authenticate(name,password) to Server
}
```

Conclusion

A proof theoretic foundation for session types:

- The first *session type system* expressed in the *calculus of structures* enabling:
    - a natural notion of **multi-party compatibility** (using provability);
    - A **consistent** notion of **subtyping** (using linear implication);
- Projection from *global types* guarantees multi-party compatiblility.

Applications to security include:

- Runtime monitoring to detect violations of specified protocols.
- Type checking code for confomance to a role in a security protocol.
- Verification of security protocols themselves in dependently typed extensions.

Future extensions include fixed points or replication to enable the analysis of protocols with unbounded participants and the behaviour of attackers with the ability to initiate unbounded sessions.

Extra Example: Tiu's Counterexample

Role P: $\sim$*begin*(*Data*) `to` Q ;
{
  `par` $\sim$*fun*(*Control*) `to` Q
  `and` *done*(*Data*) `from` Q
}

Role Q: {
  `par` *begin*(*Data*) `from` P
  `and` *fun*(*Control*) `from` P
} ;
$\sim$*done*(*Data*) `to` P

Coordinating middleware: `sync` *begin*(*Data*) `to` Q and $\sim$*begin*(*Data*) `from` P

`sync` *fun*(*Control*) `to` Q and $\sim$*fun*(*Control*) `from` P

`sync` *done*(*Data*) `to` P and $\sim$*done*(*Data*) `from` Q

Extra Example: Tiu's Counterexample

```
par {
      ~begin(Data) to Q ;
      {
          par ~fun(Control) to Q
          and done(Data) from Q
       }
}
and {
      {
          par begin(Data) from P
          and fun(Control) from P
      } ;
      ~done(Data) to P
}
and {
    sync begin(Data) to Q and ~begin(Data) from P
}
and {
    sync fun(Control) to Q and ~fun(Control) from P
}
and {
    sync done(Data) to P and ~done(Data) from Q
}
```

Extra Example: Tiu's Counterexample (deep step)

```
par {
    ∼begin(Data) to Q ;
    ∼fun(Control) to Q ;
    done(Data) from Q
     }
}
and {
    begin(Data) from P ;
    fun(Control) from P ;
    codone(Data) to P
}
and {
    sync begin(Data) to Q and ∼begin(Data) from P ;
    sync fun(Control) to Q and ∼fun(Control) from P ;
    sync done(Data) to P and ∼done(Data) from Q
}
```

Extra Example: Tiu's Counterexample

```
{
    par
       ~begin(Data) to Q
    and
       begin(Data) from P
    and
       sync begin(Data) to Q and ~begin(Data) from P
};
    par
       ~fun(Control) to Q
    and
       fun(Control) from P
    and
       sync fun(Control) to Q and ~fun(Control) from P
};
    par
       done(Data) from Q
    and
       codone(Data) to P
    and
       sync done(Data) to P and ~done(Data) from Q
 }
```

Extra Example: Tiu's Counterexample

```
{
    sync {
        par ~begin(Data) to Q and begin(Data) to Q
    }
    and {
        par ~begin(Data) from P and begin(Data) from P
    }
} ;
{
    sync {
        par ~fun(Control) to Q and fun(Control) to Q
    }
    and {
        par ~fun(Control) from P and fun(Control) from P
    }
} ;
{
    sync {
        par ~done(Data) to P and done(Data) to P
    }
    and {
        par ~done(Data) from Q and done(Data) from Q
    }
}
```

Extra Example: Tiu's Counterexample

{ }

Tiu's counterexample is coherent with respect to:

begin(*Data*) `from` *P* to *Q* ;
fun(*Function*) `from` *P* to *Q* ;
done(*Data*) `from` *Q* to *P*