A contract-oriented middleware

Security Foundations

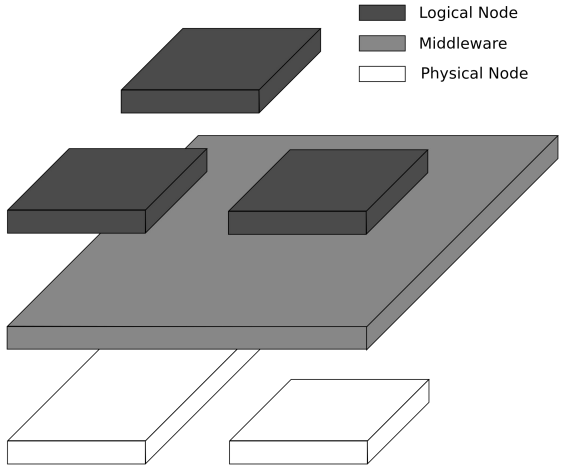
atzeinicola@gmail.com

Topics

- contract-oriented middleware
- contracts: (timed) session types
- distributed contract-oriented applications (Java API)

What is a middleware?

- a computer software
- provides services to other software applications
- makes it easier for developers to perform *communication* and I/O ops



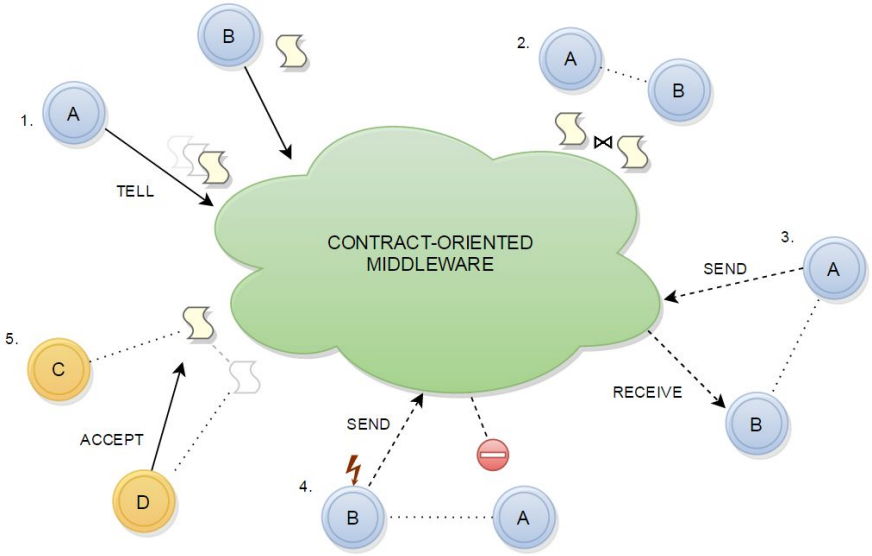
What is a contract?

A contract is an abstraction of the intended behaviour of a participant. It models both what you *offer* to others and what you *require* in exchange.

Several models:

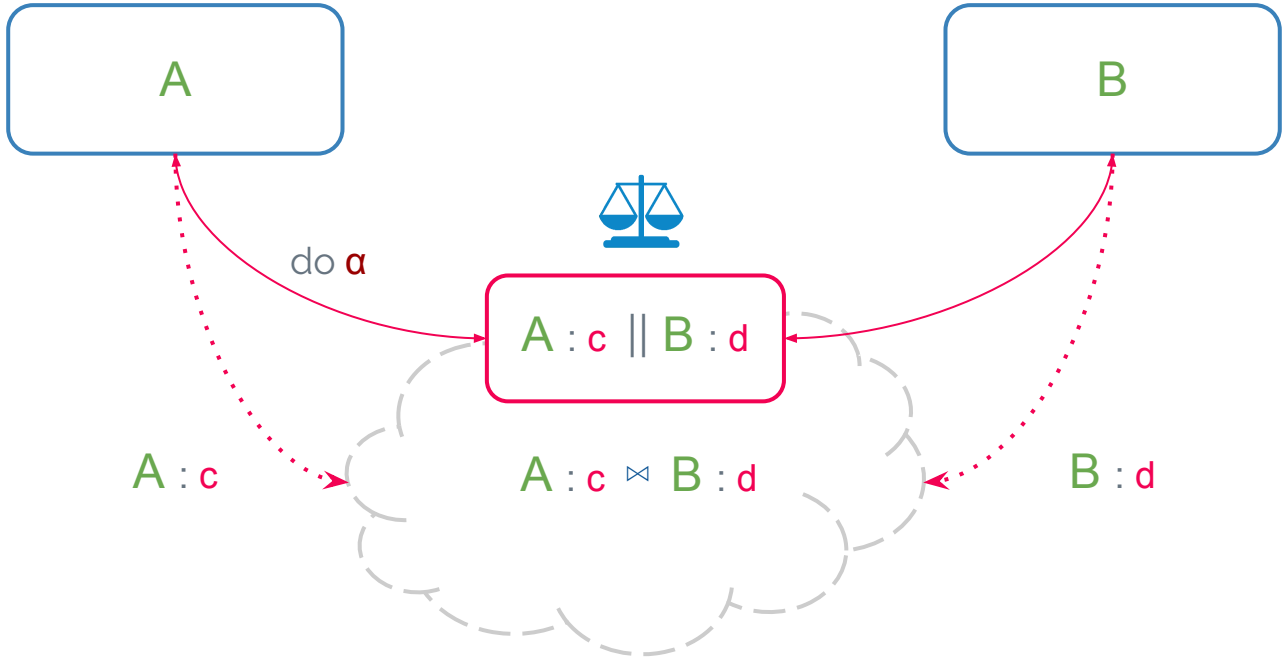
- event structures
- **session types**
- **timed session types**
- ...

What is the CO_2 middleware?



CO_2 collects *contracts* advertised by participants, and establishes *sessions* between those with *compliant* ones

Contract-oriented computing



Interaction

The interaction between two participant is monitored by the middleware.

It creates a contract configuration A : c | B : d that represents the state of a session. When a participant performs an action (send or receive), the contract configuration changes accordingly.

The middleware can inspect a contract configuration and determine who is the participant in debt with the other one, causing the session to be stuck.

Compliance

The intuition is that if a contract c is compliant with a contract d, then in *all* the configurations of a computation of A : c | B : d, whenever a participant wants to choose a branch in an internal sum, the other participant always offers the opportunity to do it.

Compliance guarantees that whenever a computation of A : c | B : d becomes stuck, then both participants have reached the success state 0

Session types

Session types

Session types are terms of a process algebra featuring internal/external choice, and recursion

$$c = ?zip \,.\, (\, !weather + !abort \,)$$

$$d = !zip \,.\, (\, ?weather \,\&\, ?abort \,)$$

- c is compliant with d (in symbols $c \bowtie d$)
- c is the dual of d (and viceversa)

Example of compliant contracts

$A : ?zip.(!weather + !abort) \mid B : !zip.(?weather \,\&\, ?abort)$

$\xrightarrow{B:\,!zip}\; A : [?zip].(!weather + !abort) \mid B : ?weather \,\&\, ?abort$

$\xrightarrow{A:\,?zip}\; A : !weather + !abort \mid B : ?weather \,\&\, ?abort$

$\xrightarrow{A:\,!weather}\; A : 0 \mid B : [?weather] \xrightarrow{B:\,?weather} A : 0 \mid B : 0$

$\xrightarrow{A:\,!abort}\; A : 0 \mid B : [?abort] \xrightarrow{B:\,?abort} A : 0 \mid B : 0$

Reminder

Compliance guarantees that whenever a computation of $A : c \mid B : d$ becomes stuck, then both participants have reached the success state 0.

Example of not compliant contracts

$A : ?zip.(!weather + !abort) \mid B : !zip.\ ?weather$

$\xrightarrow{B:\,!zip}\ A : [?zip].(!weather + !abort) \mid B : ?weather$

$\xrightarrow{A:\,?zip}\ A : !weather + !abort \mid B : ?weather$

$\xrightarrow{A:\,!weather}\ A : 0 \mid B : [?weather] \xrightarrow{B:\,?weather} A : 0 \mid B : 0$

$\xrightarrow{A:\,!abort}$

Reminder

Compliance guarantees that whenever a computation of $A : c \mid B : d$ becomes stuck, then both participants have reached the success state 0.

Timed session types

Timed session types

Timed extension of session types:

session types + clocks + guards/reset

c = ?zip {x<30;x} . (!weather {5<x<20} + !abort {x<5})

d = !zip {x<30;x} . (?weather {x<15} & ?abort)

These contracts are *not compliant*. In fact, the contract d wants to receive a weather in at most 15 sec, whereas the contract c can send it up to 20 seconds.

Examples

A		B
?a{t<5}.!b{t<3}	⋈	!a{t<2}.?b{t<3}
?a{t<5}.!b{t<3}	⋈̸	!a{t<5}.?b{t<3}
!a{x<3}.!b{x<2}	⋈̸	?a{x<3}.?b{x<2}
!a{x<2}.!b{x<2}	⋈	?a{x<3}.?b{x<2}
!a{x<3}+!b{x<2}.?a{x<1}	⋈̸	?a{x<3}&?b{x<2}.!a{x<1}
?zip{;x}.(!weather{x>5, x<10} + !abort{x<1})	⋈̸	!zip{;y}.(?weather{y<7} & ?abort{y<5})
?zip{;x}.(!weather{x>5, x<10} + !abort{x<1})	⋈̸	!zip{y<10}.(?weather{y<7} & ?abort{y<5})

Try it yourself

http://co2.unica.it/tst/compliance/

CO_2 Java API

Setup

- install groovy (http://www.groovy-lang.org/download.html)
- download the latest version of Java API at http://co2.unica.it/downloads/co2api and save it into **<home>/.groovy/lib/**
- create the file **<home>/.groovy/groovysh.rc** and insert **import co2api.***

Connect to the middleware

Create a connection to the CO_2 middleware

```
conn = new CO2ServerConnection(user, password)
```

where

```
user     = "<matricola>@co2.unica.it"
password = "<matricola>"
```

Test your connection with the following method call

```
connection.serverTime()
 => 1449563789864
```

Create a contract

Create a private contract (Timed Session Type)

```
tst = new TST(contract)
```

Alice

```
contract = "!a + !b"
```

Bob

```
contract = "?a & ?b"
```

You can check the compliance of two TST with

```
tst.isCompliantWith(tst2)
```

Advertise a contract

Advertise the contract to the middleware

```
publicContract = tst.toPrivate(conn).tell()
```

Your contract is public: the middleware has collected it and it establishes a session with another participant that advertises a compliant contract (like Alice and Bob)

Get the unique ID of your contract and check if it is fused

```
publicContract.getUniqueID()
 => b1e4b41516d9e699ed51c81a8897c456c34a...

publicContract.isFused()
 => false/true
```

Retract a contract

You can retract your contract, ensuring that the middleware will not use it to find a compliant participant

```
publicContract.retract()
 => true
```

If your contract is already involved in a session (`publicContract.isFused() == true`) the method will throw an exception

```
publicContract.retract()
 => ContractException: ...
```

Establish a session

Establish a new session

```
session = publicContract.waitForSession()
```

You can also specify a **timeout**: if it expires, the method will throw an exception

```
session = publicContract.waitForSession(10_000)
 => ContractTimeoutException: ...
```

Now your contract is fused

```
publicContract.isFused()
 => true
```

Establish a session (2)

During a session, only one participant is **on duty**

```
session.isEnded()
 => false

session.amIOnDuty()
 => true (Alice)
 => false (Bob)
```

When the session is ended, nobody is on duty and only one participant could be **culpable**

```
session.amICulpable()
```

Sending messages

A participant can send messages to the other endpoint specifing an **action** and a optional **value**

```
session.send("a")
session.send("a", "string value")
session.send("a", 42)
```

The other participant can receive a message and get its value

```
msg = session.waitForReceive()
msg.getLabel()
  => "a"
msg.getStringValue()*
msg.getIntegerValue()*
```

* it throws a `ContractException` if the type is incorrect

Context

The concept of context is used to define permitted actions and their types, and to validate the actions.

The context exists a priori, so you cannot define it using the API.

When creating a contract, you can associate it to a context as follows

```
new TST(contract, "context-name")
```

Note: if your contract contains actions that are not present in the context, an exception is thrown when trying to tell it.

Context or not?

There are some differences to consider:

Context	Without context
▷ custom validation	▷ no validation
▷ action type (int or string)	▷ no action type (always string)
▷ only contracts with permitted actions	▷ any valid contract
▷ you can send only permitted actions*	▷ you can send any action

* otherwise an exception is thrown, but it does not make you culpable

Proof of presence

Lab context

I have created a context named **lab** defined as follows:

Action label	Verification link	Type
hello	http://co2.unica.it/verifier/true.php	-
matricola	http://co2.unica.it/verifier/lab_co2_2015_verification_link.php*	int
token	http://co2.unica.it/verifier/true.php	string

* please verify that the link http://co2.unica.it/verifier/lab_co2_2015_verification_link.php?value=<matricola> returns true

Provide the proof

Prof. Bartoletti wrote some processes that advertise the following contract

```
?hello{x<60;x} . ?matricola{x<60;x} . !token{x<60}
```

When a session is established, it waits 60 sec. to receive a `hello` message, then it waits 60 sec. to receive a `matricola` message, finally it sends back a `token` within 60 sec.

Write a program in order to obtain your token!

Solution

```
username = "<matricola>@co2.unica.it"
password = "<matricola>"
connection = new CO2ServerConnection(username,password)

contract = "!hello{x<60;x}.!matricola{x<60;x}.?token{x<60}"
ctx = "lab"

pvtContract = new TST(contract, ctx).toPrivate(connection)
pblContract = pvtContract.tell()

session = pblContract.waitForSession()
session.send("hello")
session.send("matricola", "<matricola>")

msg = session.waitForReceive()
msg.getStringValue()
 => <your token>
```

References

- http://co2.unica.it/
- **A contract-oriented middleware**, M. Bartoletti, T.Cimoli, M. Murgia, A. S. Podda, L. Pompianu, 2015. *In Proc. FACS* (pdf)
- **Compliance and subtyping in timed session types**, M. Bartoletti, T.Cimoli, M. Murgia, A. S. Podda, L. Pompianu, 2015. *In Proc. FORTE* (pdf)