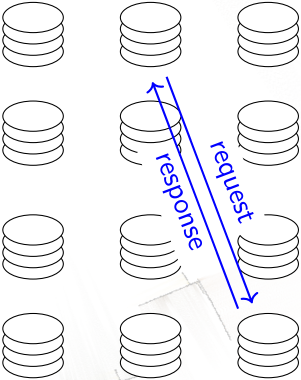
A
sound
higher order
interface description language

Joachim Breitner

September 21, 2021
Verified Systems Engineering seminar @ NUS

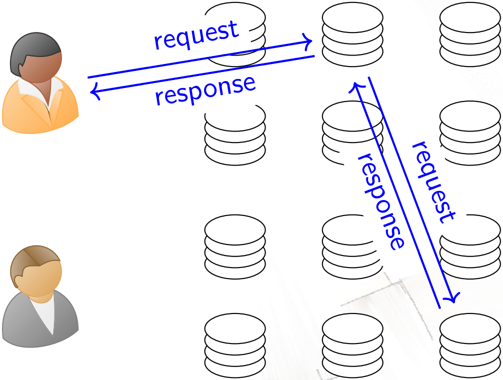
The backdrop:
DFINITY's "Internet Computer"

Internet Computer in a nutshell



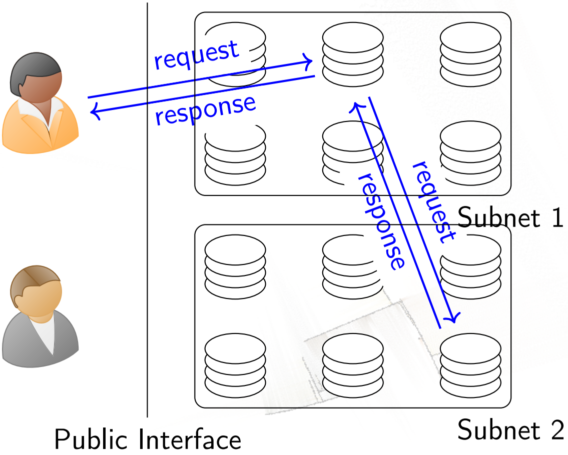
Many canisters (a.k.a. services, processes, smart contracts)

Internet Computer in a nutshell



Additionally, external users

Internet Computer in a nutshell



Different transportation layers

Internet Computer in a nutshell

The Internet Computer's system layer provides:

- Async messaging between canisters (actor model)
- Messages transport either *calls* or *responses*
- Users can perform calls and receive responses
- Payload: Method name and raw arguments (a blob)
- Canister code can be changed in general
- Some canisters are immutable ("smart contracts")

. . . not so Internet Computer specific

Our setting (in the abstract)

Our setting

- Services with identity
- Code can be upgraded
- Remote calls
- Raw data transfer

Our goals

- Describe services's interface
- Language agnostic
- Safe upgrades: interface evolution without breaking clients

How to build an IDL

Let's start with some primitive types

```
<t> ::= nat | int | float | bool | text | unit
```

... and then some composite types ...

```
<t> ::= nat | int | float | bool | text | unit
      | opt <t> | vec <t>
      | record { <name> : <t> ;∗ }
      | variant { <name> : <t> ;∗ }
```

. . . and service references (now we are higher order!)

```
<t> ::= nat | int | float | bool | text | unit
      | opt <t> | vec <t>
      | record { <name> : <t> ;∗ }
      | variant { <name> : <t> ;∗ }
      | service { <method_name> : <t> -> <t> ;∗ }
```

(Simplified for this talk; Candid has a few more and differs in some.)

Types have no value without values

```
()                          : unit
0,  1                       : nat
-42,  0,  2021              : int
true,  false                : bool
"hello"                     : text
none,  some "verse"         : opt text
[], [-1,0,1]                : vec int
{ foo = 1; bar = "baz" } : record { foo : nat, bar : text }
#foo 1,  #bar "baz"         : variant { foo : nat, bar : text }
example.service.com         : service { hello : text -> unit }
```

No communication without representation

- Define a binary wire format for all values.
  (Nothing exciting here)
- Define encoding and decoding.
  Obvious, but important: **Decoding raw bytes can fail!**
- Weird trick:
  Don't just serialize <v>, but actually <v> : <t>
  i.e. include the type at which the *sender* serialized the data.
  - May allow a more compact representation
  - Also needed for what we do next
- Oh, also integrate the IDL in the host language.
  (Left as an exercise to the reader for now)

Safe upgrades

Services want to change over time

The easy case: Additional methods

```
my_service_v1 : service {
  hello : text -> unit



}
```

↓

```
my_service_v2 : service {
  hello : text -> unit
  time_of : variant { creation; now }
                -> record { year : nat; day : nat }
}
```

Services want to change over time

The still reasonable case: Record and variant extension

```
my_service_v2 : service {
  hello : text -> unit
  time_of : variant { creation; now }
                -> record { year : nat; day : nat }
}
```

↓

```
my_service_v3 : service {
  hello : text -> unit
  time_of : variant { creation; now; birthday  : nat }
                -> record { year : nat; day : nat; seconds : nat }
}
```

The why-not case: Other compatible types

```
my_service_v3 : service {
  hello : text -> unit
  time_of : variant { creation; now; birthday : nat }
                  -> record { year : nat; day : nat; seconds : nat }
}
```

↓

```
my_service_v4 : service {
  hello : text -> unit
  time_of : variant { creation; now; birthday : int }
                  -> record { year : nat; day : nat; seconds : nat }
}
```

Services want to change over time

The no-please-no case: Changes that break clients

```
bad_service_v1 : service {
  hello : text -> unit
  weird : record { year : nat; day : nat }
                -> variant { creation; now }
}
```

$\not\downarrow$

```
bad_service_v2 : service {
  hello : text -> unit
  weird : record { year : nat; day : nat; seconds : nat }
                -> variant { creation; now; birthday : int }
}
```

This concept has a name: Subtyping!

`t1 <: t2`

subtype — supertype

Any value of *subtype* can be used at *supertype*.

This concept has a name: Subtyping!

`t1 <: t2`

subtype — supertype

supertype can evolve to subtype

Any value of *subtype* can be used at *supertype*.

Inference rules rule!

$$\frac{}{t <: t}$$

Inference rules rule!

$$\frac{}{t <: t} \qquad \frac{}{\texttt{nat} <: \texttt{int}}$$

Inference rules rule!

$$\frac{}{t \mathrel{<:} t} \qquad \frac{}{\texttt{nat} \mathrel{<:} \texttt{int}} \qquad \frac{t_1 \mathrel{<:} t_2}{\texttt{opt}\, t_1 \mathrel{<:} \texttt{opt}\, t_2} \qquad \frac{t_1 \mathrel{<:} t_2}{\texttt{vec}\, t_1 \mathrel{<:} \texttt{vec}\, t_2}$$

Inference rules rule!

$$\frac{}{t \mathrel{<:} t} \qquad \frac{}{\texttt{nat} \mathrel{<:} \texttt{int}} \qquad \frac{t_1 \mathrel{<:} t_2}{\texttt{opt}\ t_1 \mathrel{<:} \texttt{opt}\ t_2} \qquad \frac{t_1 \mathrel{<:} t_2}{\texttt{vec}\ t_1 \mathrel{<:} \texttt{vec}\ t_2}$$

$$\frac{\overline{t_1 \mathrel{<:} t_2}}{\texttt{record}\ \{\overline{n : t_1 ;}\ \overline{m : s ;}\} \mathrel{<:} \texttt{record}\ \{\overline{n : t_2 ;}\}}$$

same for service

Inference rules rule!

$$\frac{}{t \mathrel{<:} t} \qquad \frac{}{\mathtt{nat} \mathrel{<:} \mathtt{int}} \qquad \frac{t_1 \mathrel{<:} t_2}{\mathtt{opt}\ t_1 \mathrel{<:} \mathtt{opt}\ t_2} \qquad \frac{t_1 \mathrel{<:} t_2}{\mathtt{vec}\ t_1 \mathrel{<:} \mathtt{vec}\ t_2}$$

$$\frac{\overline{t_1 \mathrel{<:} t_2}}{\mathtt{record}\ \{\overline{n : t_1;}\ \overline{m : s;}\} \mathrel{<:} \mathtt{record}\ \{\overline{n : t_2;}\}}$$

same for service

$$\frac{\overline{t_1 \mathrel{<:} t_2}}{\mathtt{variant}\ \{\overline{n : t_1;}\} \mathrel{<:} \mathtt{variant}\ \{\overline{n : t_2;}\ \overline{m : s;}\}}$$

Inference rules rule!

$$\frac{}{t \mathrel{<:} t} \qquad \frac{}{\texttt{nat} \mathrel{<:} \texttt{int}} \qquad \frac{t_1 \mathrel{<:} t_2}{\texttt{opt}\ t_1 \mathrel{<:} \texttt{opt}\ t_2} \qquad \frac{t_1 \mathrel{<:} t_2}{\texttt{vec}\ t_1 \mathrel{<:} \texttt{vec}\ t_2}$$

$$\frac{\overline{t_1 \mathrel{<:} t_2}}{\texttt{record}\ \{\overline{n : t_1;}\ \overline{m : s;}\} \mathrel{<:} \texttt{record}\ \{\overline{n : t_2;}\}}$$

same for service

$$\frac{\overline{t_1 \mathrel{<:} t_2}}{\texttt{variant}\ \{\overline{n : t_1;}\} \mathrel{<:} \texttt{variant}\ \{\overline{n : t_2;}\ \overline{m : s;}\}}$$

$$\frac{t_2 \mathrel{<:} t_1 \qquad r_1 \mathrel{<:} r_2}{t_1 \texttt{ -> } r_1 \mathrel{<:} t_2 \texttt{ -> } r_2}$$

contravariance!

Subtyping ⇒ safe upgrades ≈ IDL soundness

A service can upgrade from service type

`t1` to `t2`

without breaking clients if

`t2 <: t1`

(and we can provide tools to check that)

What is IDL soundness, precisely?

What happens in a distributed system?

1. New services are added, at arbitrary type.
2. Services can begin using other services, at their current type.
3. Services can evolve, changing their type.
4. Services pass service references to other services.

It's important who believes what!

Same, but in math font

$$\frac{A \text{ fresh in } S}{S \longrightarrow (A : s) \cup S} \qquad \frac{(A : s) \in S}{S \longrightarrow (B \models A : s) \cup S}$$

$$\frac{s_1 \rightsquigarrow s_2}{\{(A : s_1), S'\} \longrightarrow \{(A : s_2), S'\}}$$

details didn't fit the slide

$$\frac{(A \models C : s_1) \in S \qquad \text{In } S,\ A \text{ can send } s_1 \text{ to } B \text{ which receives } s_2}{S \longrightarrow (B \models C : s_2) \cup S}$$

where S is a set of truths, $A \models B : s$ denotes A's belief about B's type.

The relation $\rightsquigarrow$ are the allowed service evolutions, to be instantiated with conrete rules.

The soundness criterion

I consider an Interface Definiton Language *sound*

$$\text{If } \emptyset \longrightarrow^* S,$$
$$\text{and in } S,\ A \text{ sends a message to } B,$$
$$\text{then } B \text{ can decode that message.}$$

The soundness criterion

I consider an Interface Definiton Language *sound*

$$\text{If } \emptyset \longrightarrow^* S,$$
$$\text{and in } S,\ A \text{ sends a message to } B,$$
$$\text{then } B \text{ can decode that message.}$$

This holds in general if $\rightsquigarrow$ is based on canonical subtyping.

More details in `IDL-Soundness.md` and the Coq formalization thereof.

We could be done now. . .

Unfortunately, users want to do this:

```
type User = record { name : text };
my_service : service {
  register_user : User -> unit
  find_user : text -> opt User;
}
```

↓?

```
type User = record { name : text; age : nat }
my_service : service {
  register_user : User -> unit
  find_user : text -> opt User
}
```

Maybe we can allow this?

```
type User = record { name : text }
my_service : service {
  register_user : User -> unit
  find_user : text -> opt User
}
```

↓ if missing, use none

```
type User = record { name : text; age : opt nat };
my_service : service {
  register_user : User -> unit
  find_user : text -> opt User
}
```

Subtyping for missing record optional record fields

$$\frac{}{\texttt{record}\,\{\ldots\} <: \texttt{record}\,\{n\!: \texttt{opt}\; t; \ldots\}}$$

In words: treat a missing field of type `opt` t as none.

Unfortunately, this is not sound!

```
type User = record { name : text }
type reg_service = service { register_user : User -> unit }
meta_service = service { add_listener : reg_service -> unit }
```

↓

```
type User = record { name : text;
                     age : opt variant { child; adult } }
```

At the old types,

```
meta_service.add_listener(my_service)
```

is well typed.

But after upgrades,

`meta_service` sends `opt variant { child; adult }` but

`my_service` expects `opt nat`. ← BOOM

To fix that, opt is special

$$\frac{}{\texttt{opt}\ t_1 \texttt{ <: } \texttt{opt}\ t_2}$$

look, no assumptions!

When decoding, check given type t_1 against expected type t_2:

- If t_1 `<:` t_2, use the value,
- else, ignore value, treat as none

This is a dynamic type check!

And while we are at it...

$$\frac{}{t_1 \texttt{ <: opt } t_2}$$

any type works! →

When decoding, if t_1 is not an `opt`..., pretend it is, and continue as before.

And while we are at it. . .

any type works! →

$$\frac{}{t_1 \texttt{ <: opt } t_2}$$

When decoding, if t_1 is not an `opt` . . ., pretend it is, and continue as before.

Use case: Previously required arguments can be made optional.

And while we are at it...

$$\frac{}{t_1 \texttt{ <: opt } t_2}$$

any type works! →

When decoding, if t_1 is not an `opt` ..., pretend it is, and continue as before.

Use case: Previously required arguments can be made optional.

Additional complexities with equirecursive types (`opt opt opt` ...)
Better restrict this to only when t_2 is itself not an opt type.

Alternatives?

- Can one really not avoid the dynamic check?
  We considered special *argument record* types, or special field markers, that change subtyping to allow extension in argument position. But breaks using the same type definitions in argument and result position.
- Is it maybe enough to dynamically check the *value*?
  No: service reference values would slip through, breaking soundness.
- One can at least use a dedicated type operator (upgraded...)?
  Yes, that works
- Any other weird ideas?
  Plenty. See Motoko issue #1523 for the full epic saga.

Summary

- A interface description language is important for distributed systems
- We defined what sound and higher order means
- Canonical subtyping does what we want, in general
- Record extension in both positions is possible, but tricky
- We skipped a bunch of (mostly) engineering decisions

Thank you for your attention!

Further reading:

- The Candid spec
- The Candid manual
- My Candid explainer blog post
- The IDL Soundness definition
- The Coq formalization