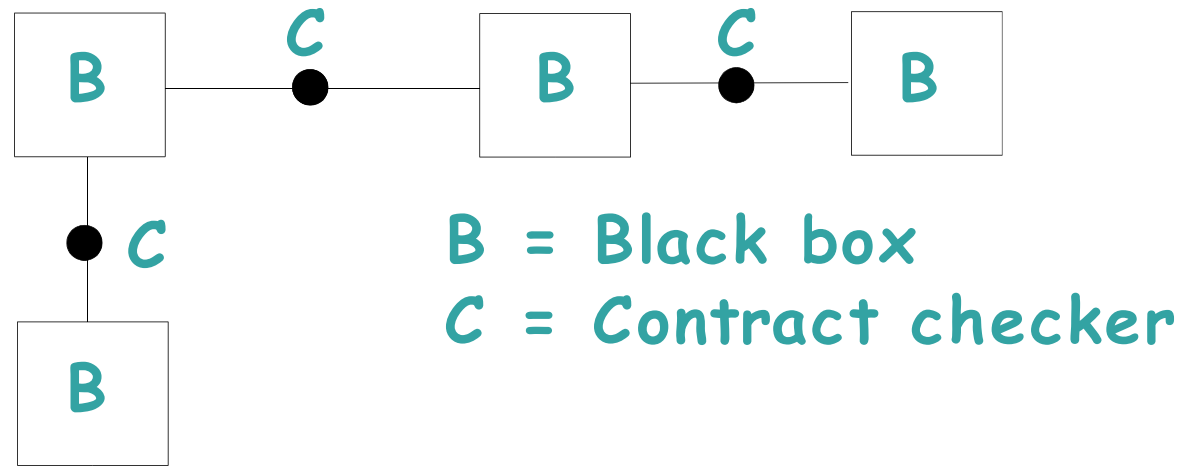


UBF

Joe Armstrong

joe@sics.se

UBF 1



Components in a distributed system communicate by following some protocol.

Question: How can we ensure that all components in a distributed system obey the rules of the protocol?

Answer: Contracts.

Question: How should programs written in different languages talk to each other?

Answer: UBF.

How do we make systems?



Systems are made of black boxes
(components)

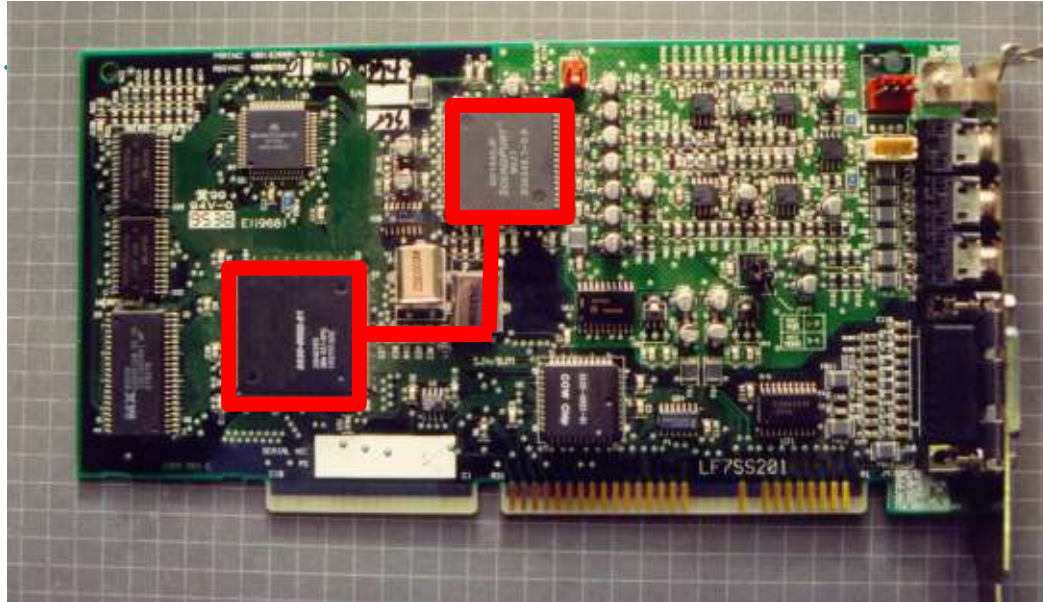
Black boxes execute concurrently

Black boxes communicate

How the black box works internally is
irrelevant

Failures inside one black box cannot crash
another black box

That's how hardware works



Hardware components operate concurrently are isolated and communicate by message passing

Remember slide 5?

Universal Binary Format

UBF(A) = An language independent encoding scheme.

Approximately equivalent to well-formed XML - but much more efficient.

UBF(B) = A type scheme and a contract language.

Approximately equivalent to XML schemas + WSDL but more expressive.

UBF(A)

Primitive types

Integers 3987597834 -348576 ...
strings "this is a string"
memory buffers <Int>~~
constants `monday` `november` etc.

Constructed types:

tuples {Arg1,Arg2,...,Argn}
lists #Arg1&Args2&...&Arg

White space (tab, nl, blanks) is ignored.

>C pop the top item of the recognition stack
 and store into reg[C]
C push reg[C] onto the recognition stack

UBF(A) - example

```
<people>
```

```
  <person>
```

```
    <firstname>jim</firstname>
```

```
    <lastname>smith</lastname>
```

```
    <sex>male</sex>
```

```
    <age>10</age>
```

```
  </person>
```

```
  <person>
```

```
    <firstname>susan</firstname>
```

```
    <lastname>jones</lastname>
```

```
    <sex>female</sex>
```

```
    <age>14</age>
```

```
  </person>
```

```
</people>
```

```
'person'>p'
```

```
#{p,"susan","jones",'female',14}&
```

```
{p,"jim","smith",'male',10}&$
```

UBF(B) - Types

Primitive types:

int() means a UBF(A) integer
string() means a UBF(A) string
constant() means a UBF(A) constant
bin() means a UBF(A) memory buffer

X() means an object of type X

Primitive UBF(A) literals are also types.

Constructed Types:

{T1, T2, ..., Tn} is a tuple type if T1 ..Tn are types
{X1, X2, ..., Xn} is of type {T1, T2, ..., Tn} if
X1 is of type T1 and,
X2 is of type T2 and, ...
Xn is of type Tn.

[T] is the type list[T]
if T is a type.

X1 & X2 &... is of type T if
X1 is of type T and
X2 is of type T ...

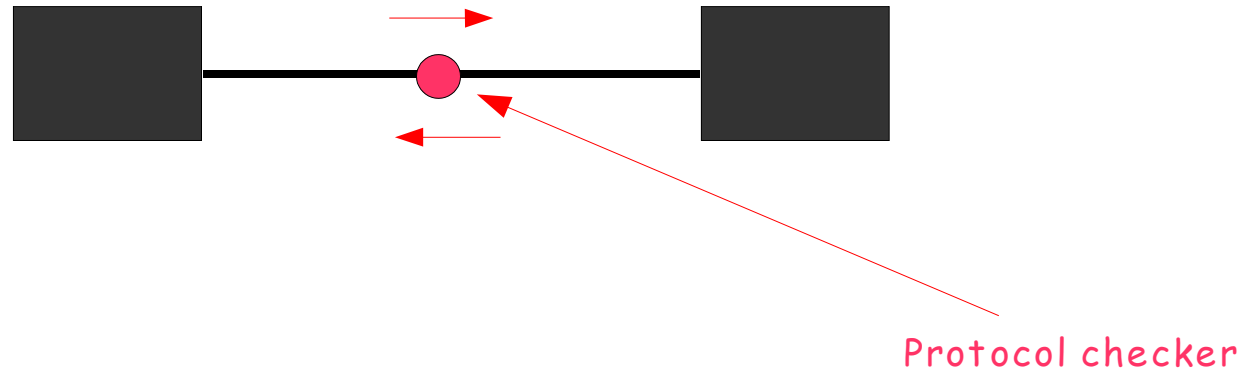
UBF(B) - Type example

+TYPES

```
person() =  
  {person,  
    firstname(),  
    lastname(),  
    sex(),  
    age()};  
firstname() = string();  
lastname() = string();  
age() = int();  
sex() = male | female;  
people() = [person()].
```

```
'person'>p'  
#{p,"susan","jones",'female',14}  
&  
{p,"jim","smith",'male',10}&$
```

How do we fit the bits together?



Systems are made of black boxes (components)

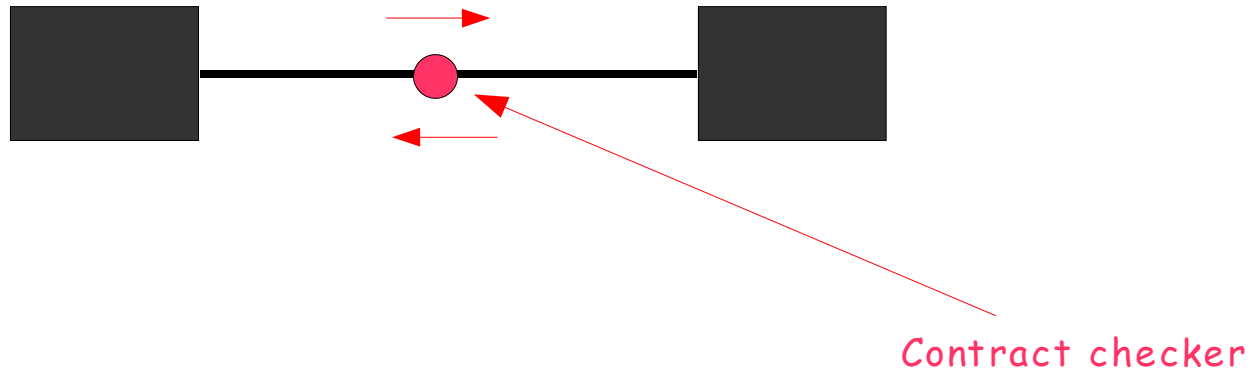
Black boxes execute concurrently

Black boxes communicate with defined (universal) protocols

The protocol is checked externally

How the black box works internally is irrelevant

Contracts



A contract is a set of four tuples of the form:

$S_{in} \times T_{in} \rightarrow S_{out} \times T_{out}$,

This means that if the server is in state S_{in} and it receives a message of type T_{in} then it may possibly respond with a message of type T_{out} and change its state to S_{out}

File server contract

+TYPES

info() = info;

description() = description;

services() = services;

contract() = contract;

file() = string();

ls() = ls;

files() = {files, [file()]};

getFile() = {get, file()};

noSuchFile() = noSuchFile.

+STATE start

ls() => files() & start;

getFile() => binary() & start

| noSuchFile() & stop.

+ANYSSTATE

info() => string();

description() => string();

contract() => term().

File server dialogue

List the files

```
'ls'$  
{{ 'files', # "ubf.erl" &  
"client.erl" &  
"Makefile" & ...
```

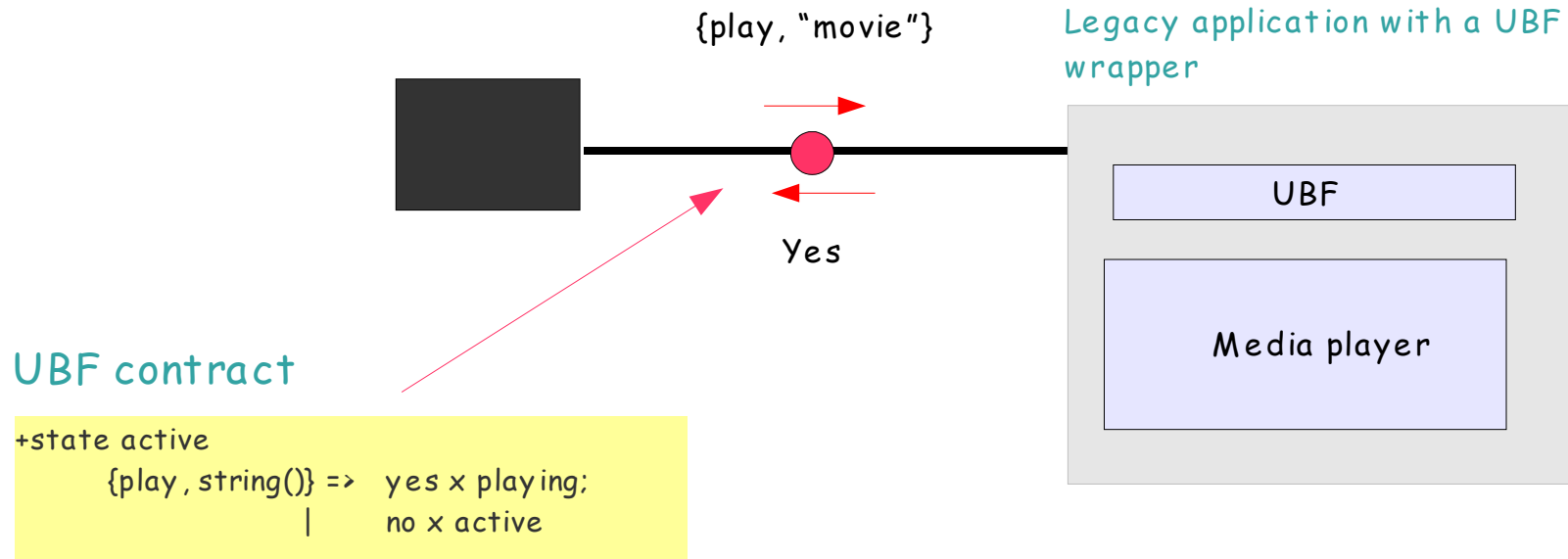
Fetch a file

```
{ 'get', "Makefile" }$  
{ 1274 ~ ... ~ 'start' }$
```

Fetch the contract - this is the UBF(A) parse tree of the UBF(B) contract given above :-)

```
'contract'$  
{ 'contract',  
  { { 'name', "file_server" },  
    { 'info', "I am a mini file server" },  
    { 'description', "  
Commands:  
  
'ls'$ List files  
{ 'get' File } => Length ~ ... ~ | noSuchFile  
  
"},  
{ 'services', # },  
{ 'states',  
  # { 'start',  
    # { 'input', { 'tuple', # { 'prim', 'file' } & { 'constant', 'get' } & },  
    # { 'output', { 'constant', 'noSuchFile' }, 'stop' } &  
      { 'output', { 'prim', 'binary' }, 'start' } & } &  
    { 'input', { 'constant', 'ls' },  
    # { 'output',  
{ 'tuple',  
  # { 'list', { 'prim', 'string' } } &  
{ 'constant', 'files' } & }, 'start' } & } & } & },  
  { 'types',  
  # { 'file', { 'prim', 'string' } } & ..... } } }$
```

Putting things together



Erlang with !!

```
"/dev/videoplayer" !!
{play, "movie"}
```

Now you see why I wanted !!

There is no `videoplayer:play(Movie)`

I want to expose the interface to the programmer

What's left to do?

Write loads of wrappers for legacy applications

Implement UBF in all languages known to man

Add non functional behaviour

- end -