Sun XDR

- **"External Data Representation"**
  - Describes serialized byte streams:

```
struct message {
  int opcode;
  opaque cookie[8];
  string name<255>;
};
```

  - Streams can be passed across the network
- **Compilers compile XDR spec to C, C++, etc.**
  - Converts messages to native data structures
  - Generates stub routines to convert struct $\leftrightarrow$ byte stream
- **Libasync `rpcc` compiles to C++**

Basic data types

- `int var` **– 32-bit signed integer**
  - wire rep: big endian (0x11223344 → 0x11, 0x22, 0x33, 0x44)
  - rpcc rep: `int32_t var`
- `hyper var` **– 64-bit signed integer**
  - wire rep: big endian
  - rpcc rep: `int64_t var`
- `unsigned int var`, `unsigned hyper var`
  - wire rep: same as signed
  - rpcc rep: `u_int32_t var`, `u_int64_t var`

More basic types

- `void` – **No data**
  - wire rep: 0 bytes of data
- `enum {name = constant,...}` – **enumeration**
  - wire rep: Same as `int`
  - rpcc rep: `enum`
- `bool var` – **boolean**
  - both reps: As if `enum bool {FALSE = 0, TRUE = 1} var`

Opaque data

- `opaque var[n]` – `n` **bytes of opaque data**
  - wire rep: `n` bytes of data, 0-padded to multiple of 4
    `opaque v[5]` → `v[0]`, `v[1]`, `v[2]`, `v[3]`, `v[4]`, 0, 0, 0
  - rpcc rep: `rpc_opaque<n> var`
    - `var[i]: char &` – `i`th byte
    - `var.size (): size_t` – number of bytes (i.e. `n`)
    - `var.base (): char *` – address of first byte
    - `var.lim (): char *` – one past last

Variable length opaque data

- `opaque var<n>` **– 0–`n` bytes of opaque data**
  - wire rep: data size in big endian format, followed by `n` bytes of data, 0-padded to multiple of 4
  - rpcc rep: `rpc_bytes<n> var`
    - `var.setsize (size_t n)` – set size to `n` (destructive)
    - `var[i]: char &` – `i`th byte
    - `var.size (): size_t` – number of bytes
    - `var.base (): char *` – address of first byte
    - `var.lim (): char *` – one past last
- `opaque var<>` **– arbitrary length opaque data**
  - wire rep: same
  - rpcc rep: `rpc_bytes<RPC_INFINITY> var`

Strings

- `string var<n>` **– string of up to** `n` **bytes**
  - wire rep: just like `opaque var<n>`
  - rpcc rep: `rpc_str<n>` behaves like `str`, except cannot be NULL, cannot be longer than `n` bytes
- `string var<>` **– arbitrary length string**
  - wire rep: same as `string var<n>`
  - rpcc rep: same as `string var<RPC_INFINITY>`
- **Note: Strings cannot contain 0-valued bytes**
  - Should be allowed by RFC
  - Because of C string implementations, does not work
  - rpcc preserves "broken" semantics of C applications

Arrays

- `obj_t var[n]` – **Array of** `n obj_t`**s**
  - wire rep: `n` wire reps of `obj_t` in a row
  - rpcc rep: `array<obj_t, n> var`; as for opaque: `var[i]`, `var.size ()`, `var.base ()`, `var.lim ()`
- `obj_t var<n>` – **0–**`n obj_t`**'s**
  - wire rep: array size in big endian, followed by that many wire reps of `obj_t`
  - rpcc rep: `rpc_vec<obj_t, n> var`; `var.setsize (n)`, `var[i]`, `var.size ()`, `var.base ()`, `var.lim ()`

Pointers

- `obj_t *var` – **"optional"** `obj_t`
  - wire rep: same as `obj_t var<1>`: Either just 0, or 1 followed by wire rep of `obj_t`
  - rpcc rep: `rpc_ptr<obj_t> var`
    - `var.alloc ()` – makes `var` behave like `obj_t *`
    - `var.free ()` – makes `var` behave like NULL
    - `var = var2` – Makes a copy of `*var2` if non-NULL
- **Pointers allow linked lists:**

```
struct entry {
  filename name;
  entry *nextentry;
};
```

- **Not to be confused with network object pointers!**

Structures

```
struct type {
  type_A fieldA;
  type_B fieldB;
  ...
};
```

- **wire rep: wire representation of each field in order**
- **rpcc rep: structure as defined**

Discriminated unions

```
union type switch (simple_type which) {
  case value_A:
    type_A varA;
  ...
  default:
    void;
};
```

- `simple_type` **must be** `[unsigned] int`, `bool`, **or** `enum`
- **Wire representation: wire rep of** `which`**, followed by wire rep of case selected by** `which`**.**

Discriminated unions: rpcc representation

```
struct type {
  simple_type which;
  union {
    union_entry<type_A> varA;
    ...
  };
};
```

- `void type::set_which (simple_type newwhich)` **sets the value of the discriminant**
- `varA` **behaves like** `type_A *` **if** `which == value_A`
- **Otherwise, accessing** `varA` **causes core dump (when using dmalloc)**

Example: fetch and add server

```
struct fadd_arg {
  string var<>;
  int inc;
};

union fadd_res switch (int error) {
case 0:
  int sum;
default:
  void;
};
```

RPC program definition

```
program FADD_PROG {
  version FADD_VERS {
    void FADDPROC_NULL (void) = 0;
    fadd_res FADDPROC_FADD (fadd_arg) = 1;
  } = 1;
} = 300001;
```

Client code

```
fadd_arg arg; fadd_res res;

void getres (clnt_stat err) {
  if (err) warn << "server: " << err << "\n";  // pretty-prints
  else if (res.error) warn << "error #" << res.error << "\n";
  else warn << "sum is " << *res.sum << "\n";
}

void start () {
  int fd;
  /* ... connect fd to server, fill in arg ... */
  ref<axprt> x = axprt_stream::alloc (fd);
  ref<aclnt> c = aclnt::alloc (x, fadd_prog_1);
  c->call (FADDPROC_FADD, &arg, &res, wrap (getres));
}
```

Server code

```
qhash<str, int> table;

void dofadd (fadd_arg *arg, fad_res *res) {
  int *valp = table[arg->var];
  if (valp) {
    res.set_error (0);
    *res->sum = *valp += arg->inc;
  } else
    res.set_error (NOTFOUND);
}

void getnewclient (int fd) {
  asrv::alloc (axprt_stream::alloc (fd), fadd_prog_1,
               wrap (dispatch));
}
```

Server dispatch code

```
void dispatch (svccb *sbp) {
  switch (sbp->proc ()) {
  case FADDPROC_NULL:
    sbp->reply (NULL);
    break;
  case FADDPROC_FADD:
    fadd_res res;
    dofadd (sbp->template getarg<fadd_arg> (), &res);
    sbp->reply (&res);
    break;
  default:
    sbp->reject (PROC_UNAVAIL);
  }
}
```