An Open Modular Middleware for Interoperable Virtual Environments

Sergiy Byelozyorov
Overview

Why?

What?

How?
Why Do We Need a New Middleware?

● Approaches
  ○ Fixed wire protocols
    ■ fragile and hard to extend
  ○ Middleware solutions
    ■ need to copy data, or
    ■ use middle-specific data structs in the application
    ■ do not allow efficient memory-mapped communication mechanism (RDMA)
Existing Virtual World Protocols

● Second Life Protocol
  ○ implemented in application logic
  ○ may be extended by Capabilities
Existing Virtual World Protocols

- MPEG-M
  - originally built for IPTV
  - suitable for building virtual worlds
  - limited to the engines specified in the standards
Existing Virtual World Protocols

- Sirikata Protocol
  - synchronizes time and objects
  - delivers generic object-to-object messages
  - no mechanism for discovery of supported extensions
What Do We Offer?

- **KIARA**
  - Separates three concerns in middleware systems:
    - the IDL or **what** needs to be transferred,
    - the API or **how** the application makes the data available,
    - protocols and transport mechanisms.
  - Allows to negotiate the exact protocol at runtime
    - based on various requirements and available technology
  - Application describes the layout its data structures
    - may use reflection in high-level languages like C# and Java
What Do We Offer?

● KIARA

○ Uses an embedded compiler to create optimized code
  ■ for accessing application data
  ■ formatting it as necessary for the negotiated communication mechanism

○ May interact with non-KIARA based systems
  ■ requires to implement this protocol in KIARA
What Do We Offer?

- Modular protocol
  - Only *interface query mechanism* is required
  - Other services are queried via this mechanism
  - Enables to write universal client and server software

```java
service kiara {
    boolean[] implements (string[] interfaceURIs);
}
```
How Can It Be Useful?

Demo
Thank you
Second Life Protocol

Header (Packet ID)

- **LL_ZERO_CODE_FLAG 0x80** - 0's in packet body are run length encoded, such that series of 1 to 255 zero bytes are encoded to take 2 bytes.
- **LL_RELIABLE_FLAG 0x40** - This packet was sent reliably (implies please ack this packet)
- **LL_RESENT_FLAG 0x20** - This packet is a resend from the source.
- **LL_ACK_FLAG 0x10** - This packet contains appended acks.
- **Byte 0**: 4 least significant bits: currently unused
- **Bytes 1-4**: Sequence number as a big-endian unsigned integer
- **Byte 5**: How many bytes of extra header is available

If byte 5 is non-zero, then there is some extra header information. Clients which are not expecting that header information may skip it by jumping forward ‘Extra’. The term ‘Packet ID’ is sometimes used to refer to all 4 bytes and sometimes just the sequence number.

For further discussion of the bits in the header, see Messages and Packet Accounting.

Body

- **Bytes 6+extra to 6+extra+(data length)**
  - Message number. This is a numeric encoding of the message types defined in the message_template.msg file. It may be 1, 2 or 4 bytes in length, depending on frequencies.
    - High frequency messages are assigned (from the template file) numbers 0x01 - 0xFE.
    - Medium frequency messages are assigned (derived from numbers in the template file) numbers 0xFF01 - 0xFFFE.
    - Low frequency messages are assigned (derived from numbers in the template file) numbers 0xFFFF0001 and up.
  - Messages with “Fixed” frequency are really those with fixed message numbers, i.e. the numbers are assigned in the message_template.msg file itself