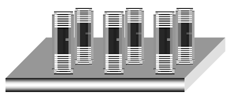
Middleware Interoperability in SOA Applications

J. Alan Brown,

www.iona.com/artix

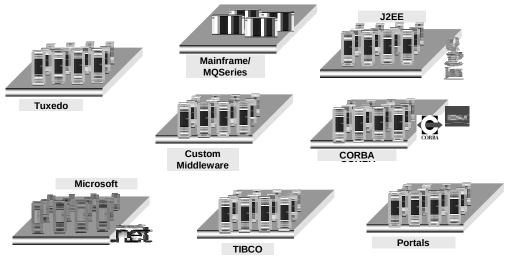
March 2004

The nice thing about middleware is that it integrates applications



Middleware choices: CORBA, J2EE, Tuxedo, MQSeries,….

The bad thing is that different middleware doesn't interoperate



This results in "islands of integration". Ad-hoc techniques can be used to integrate these, but none of these are really satisfactory; and most enterprises are too busy to research how best to do their own integrations.

More than one middleware: why?

Here are some of the reasons:

- Historical reasons in the enterprise
  - Middleware products have been on sale for many years
- Sophisticated users:
  - Select middleware that best matches each integration need
    - Synchronous request-reply (possibly RPC-based API)
    - Asynchronous messages (possibly fixed API)
    - Pub-sub
    - EAI (messages, transformation, routing)
    - QoS trade-offs: high-throughput, logging, transactions, security, …
- Each department/division of a large enterprise may make its own decisions
- Some of the applications that the enterprise has purchased will have introduced new middleware

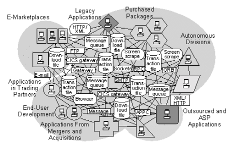
We don't believe that this is a passing phenomenon

- Different middleware has different advantages
- Departments may still insist on making their own decisions
- A "super-middleware" is unlikely
  - It's such a big technical area, that a "super-middleware" would be unwieldy
  - Some of the properties are mutually exclusive
- The cost of changing to new technology is high
  - And you may be half way there when the next new thing arrives

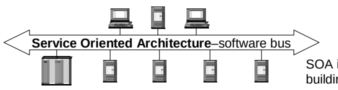
Service Oriented Architecture

- Let's start with an "ideal solution" to integrate the islands

Intead of ad-hoc integration:



…let's assume that all applications provide **services** that can be used by all others (that have security clearance) – even across the "islands of middleware":

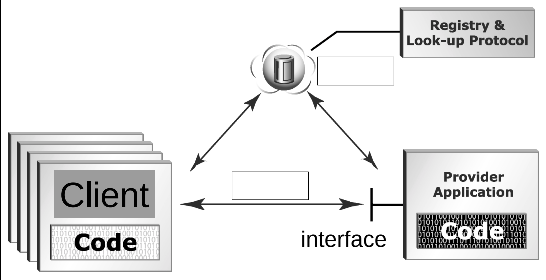


SOA is a set of principles for building systems. It helps you to master complexity.

Services

- Each service has a well-defined interface
  - Listing the operations it provides (or you may view this as the set of messages it accepts and sends in response to requests)
- SOA works well at the business level
  - The services are high-level entities meaningful to the business
- It also offers much to the technical level
  - The services are meaningful to the implementer (objects, components)
  - A business level service can be implemented by multiple objects/components, or by a non-OO approach
- SOA is not a new idea, but it's very much in vogue today
  - Analysts recommend that companies move towards it

SOA – zoom in on one interaction



SOA and middleware

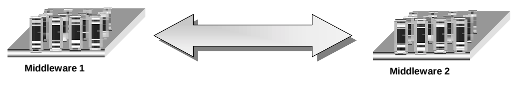
- SOA can be achieved with many types of middleware
  - But some provide direct support
    - The middleware works in terms of services
  - Any MoM has the basics in place for SOA
  - CORBA and EJBs are closer still
    - They provide native interface definition languages
  - EAI could struggle
  - Web Services offers direct support
    - And maybe it's good that we have a SOA-middleware that is aimed at a higher level than CORBA and EJBs
      - This may help to keep the business and implementation levels separate for the designers and implementers

The SOA bus

- This is a complex piece of software:
  - Quality of Service
    - Security
    - Transactions
    - Routing
    - Logging
    - SNMP message generation
    - Failover
    - …
    - And of course: protocols, message formats, …
  - Which brings us back to reality. How can a SOA bus integrate the software islands?

    SOA is the principle that we want to aim for, but it's not a solution until we can solve the "islands of integration" problem. Put another way: we want an **enterprise-wide SOA**.

To get an enterprise-wide SOA, we need to have middleware interoperability



- But middleware doesn't interoperate
- Even the "service" interfaces are very different
  - A services has an interface, and this is the key to hiding details such as the operating system, programming language, network addresses, etc
  - But this interface doesn't hide/abstract the middleware itself

Some examples…

A IDL Interface ⇒ CORBA

```
typedef  float   price;
struct place {char row; long seat;};

typedef  sequence<place>  places;

interface FrontOffice  {
  readonly attribute string  name;
  places checkAvailability();
  price checkPrice(in place p);
  boolean bookSingle(in place choice,
                     in string creditCard);
  . . .
};
```

A MoM Interface – meaningless in itself

Get the name

Input message

Function Identifier : integer = 1

Output message

string

Check availability

Input message

Function Identifier : integer = 2

Output message

Numer of places : integer
row 1 : char
seat 1 : integer
row 2 : char
seat 2 : integer
:
row n : char
seat n : integer

../cont

Get the price

Input message

Function Identifier : integer = 3
row : char
seat : integer

Output message

float

Book a single seat

Input message

Function Identifier : integer = 4
row : char
seat : integer
credit card : string

Output message

boolean

Some MoM use a programming language to define the set of messages

- For example, MQ programmers often use COBOL

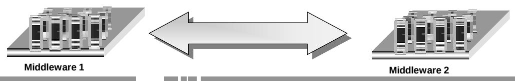
```
01 CUSTOMER-SEARCH.
      05 FIRST-NAME PIC X(25).
      05 LAST-NAME PIC X(25).
```

```
01 CUSTOMER-SEARCH-RESULTS.
   05 THE-STATUS.
      10 RESULT-CODE PIC 9(5).
      10 THE-MESSAGE PIC X(25).
   05 RESULT-COUNT PIC 9(2).
   05 CUSTOMER-RECORDS OCCURS 100 TIMES.
      10 FIRST-NAME PIC X(25).
      10 LAST-NAME PIC X(25).
      10 THE-ADDRESS.
         15 STREET-ADDRESS PIC X(50).
         15 CITY PIC X(25).
         15 STATE PIC X(25).
         15 ZIP-CODE PIC X(25).
```

Or they may just use some pages typed into a word processor to define the messages, with or without diagrams to help.

The bottom line:

- The interface doesn't abstract the chosen middleware
  - Once you see the interface(s), you know what middleware you <u>must</u> use to talk to my service
    - And in practice, you know what protocols you must use as well.
      - Because each middleware supports only a small selection.
- So middleware doesn't even attempt to solve the problems of integrating the islands



Three approaches to addressing this

How do you integrate two islands that use different middleware?

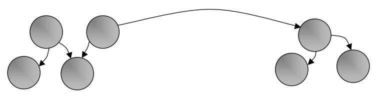
1. Use the middleware that the other side is using
2. Agree a middleware for inter-island communication
3. Use a middleware virtual switch

Really 4 approaches:

Side 2 changes \ Side 1 changes	Yes	No
No		
Yes		

How do you integrate two islands that use different middleware?

1 By one side using the middleware of the other

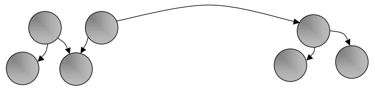


- We're not going to discuss this approach in any detail. Such compromise doesn't appear to be appropriate in most cases, unless the chosen middleware is special in some way.

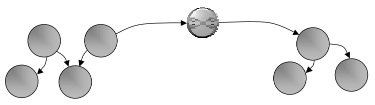
../cont.

How do you integrate two islands that use different middleware?

2 By putting a middleware in the middle



3 Using a middleware switch / translator



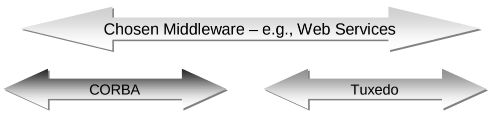
2 Middleware in the middle

- This is a valid approach
- But not without some interesting issues, such as
  - Which middleware? (Web Services for now, at least)
    - Would we ever agree?
      - Would the big computer companies agree?
    - What if new research (or standards) result in better middleware?
  - So you have 3 middlewares in use, and now you have to introduce a 4th in order to get the existing 3 to interoperate
    - Will you need more later as new middleware arrives?
  - This has to be done without requiring a *super-set middleware*
    - i.e., a "bloated" middleware that tries to achieve too much (including conflicting goals) and will not do any of these well
  - Each side must be changed to use this new middleware
    - Or provide a switch from source middleware to canonical middleware to target middleware
      - Implies double translation

../cont

Middleware in the middle

- This can result in a *nesting of layers*



Each layer exists for a different reason – and uses different middleware to achieve it's results. For example, the communication that occurs at the CORBA and Tuxedo layers may be far more tightly integrated than that of the upper layer.

3 Use a middleware virtual switch / translator

- This is a valid approach
- The number of middleware types isn't that large, so a lot of effort can be applied to integrating these
  - Working on issues such as security propagation, transactional integration, synch/asynch mapping, . . .
- New middleware can be incorporated as it emerges in the future
  - Some enterprises can extend the switch to incorporate their custum-built middleware
- The switch can do direct translation from source to target middleware, so performance can be high
  - No need for an intermediary canonical middleware

../cont

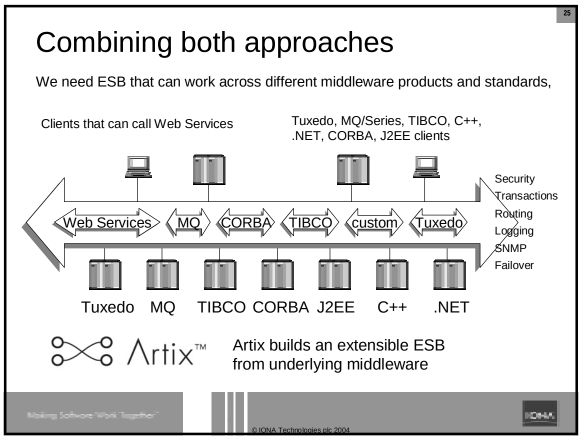
Use a middleware virtual switch / translator

- This is not the same as putting a middleware in the middle
  - no new protocols, no new wire formats
- It is related to the notion of *an agreed interface format*
  - Because the switch translates all interfaces into a canonical format so that it can determine how to translate between them.
    - But this is an implementation detail
    - The canonical interfaces are not used by the clients / servers.

Which approach to use…

- Middleware in the middle
  - Can the source and target applications be changed to use the agreed middleware?
    - Can they be changed at all?
    - Can they handle their existing and the new middleware, without the combination being too difficult?
- Middleware virtual switch
  - Less invasive
    - Because it can be used where a new middleware can't be introduced
  - But it does require translation of middleware requests (albeit that this can be kept to a single direct translation)

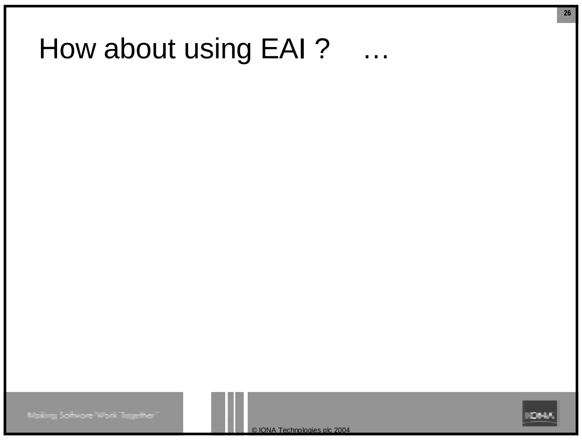
- Best: allow the two approaches to be combined



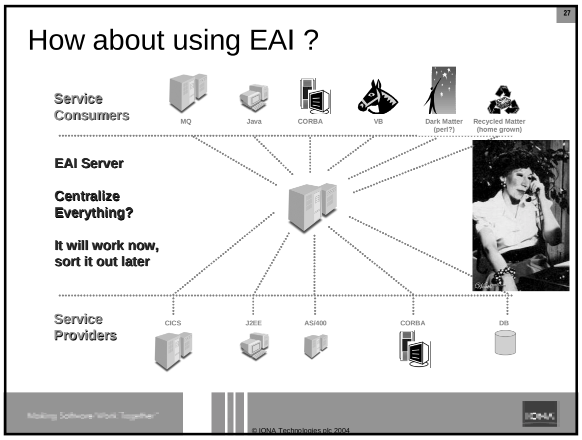
Combining both approaches

We need ESB that can work across different middleware products and standards,

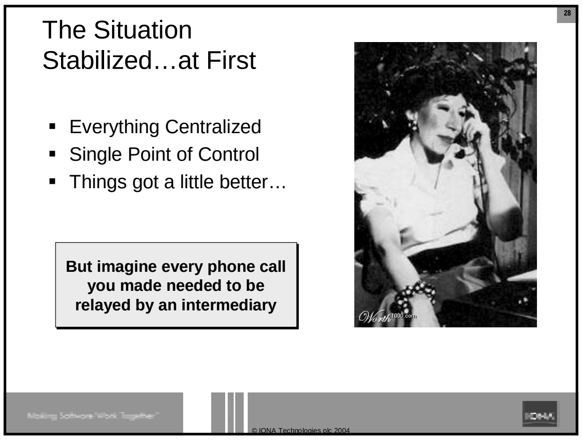
Artix builds an extensible ESB from underlying middleware



How about using EAI ? …



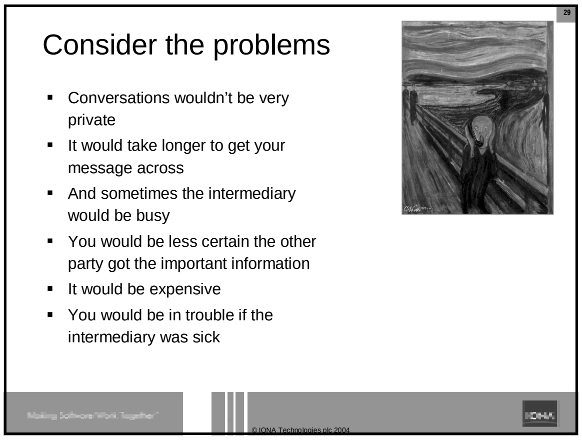
How about using EAI ?



The Situation Stabilized…at First

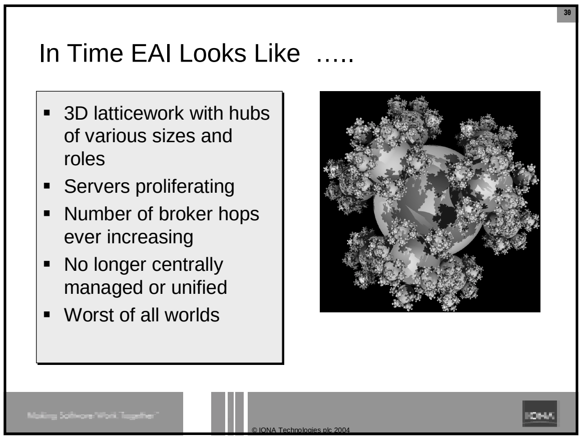
- Everything Centralized
- Single Point of Control
- Things got a little better…

But imagine every phone call you made needed to be relayed by an intermediary



Consider the problems

- Conversations wouldn't be very private
- It would take longer to get your message across
- And sometimes the intermediary would be busy
- You would be less certain the other party got the important information
- It would be expensive
- You would be in trouble if the intermediary was sick



In Time EAI Looks Like …..

- 3D latticework with hubs of various sizes and roles
- Servers proliferating
- Number of broker hops ever increasing
- No longer centrally managed or unified
- Worst of all worlds

How about other middleware … ?

- MQ
- J2EE
- Tuxedo
- TIBCO
- Others

None of these spend any effort addressing the middleware interoperability issue

➢ OK, OK, J2EE does offer JCA

- Web services

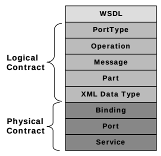
Part of any modern approach to SOA/ESB, but not a full solution.

- What if you need to get MQ and Tuxedo to talk: sometimes you will need a direct translation.

More details of using a middleware virtual switch

More details of using a middleware virtual switch

- Recall that WSDL has logical and physical parts



- Recall also the MQ interface (as defined here using COBOL)

```
01 CUSTOMER-SEARCH.
       05 FIRST-NAME PIC X(25).
       05 LAST-NAME PIC X(25).
```

```
01 CUSTOMER-SEARCH-RESULTS.
   05 THE-STATUS.
      10 RESULT-CODE PIC 9(5).
      10 THE-MESSAGE PIC X(25).
   05 RESULT-COUNT PIC 9(2).
   05 CUSTOMER-RECORDS OCCURS 100 TIMES.
      10 FIRST-NAME PIC X(25).
      10 LAST-NAME PIC X(25).
      10 THE-ADDRESS.
         15 STREET-ADDRESS PIC X(50).
         15 CITY PIC X(25).
         15 STATE PIC X(25).
         15 ZIP-CODE PIC X(25).
```

../cont

- The MQ interface can be described in WSDL

```
<binding name=”SearchFixedBinding” type=”tns:CustomerService”>
  <fixed:binding/>
  <operation name=”NameSearch”>
    <fixed:operation discriminator=”discriminator”/>
    <input>
      <fixed:body>
        <fixed:field name=”FirstName” size=”25”/>
        <fixed:field name=”LastName” size=”25”/>
      </fixed:body>
    </input>
```

../cont

```
    <output>
      <fixed:body>
        <fixed:sequence name=”return”>
          <fixed:sequence name=”item” occurs=”100”>
            <fixed:field name=”FirstName” size=”25”/>
            <fixed:field name=”LastName” size=”25”/>
            <fixed:field name=”Street” size=”50”/>
            <fixed:field name=”City” size=”25”/>
            <fixed:field name=”State” size=”25”/>
            <fixed:field name=”ZIP” size=”25”/>
          </fixed:sequence>
        </fixed:sequence>
      </fixed:body>
    </output>
  </operation>
</binding>
```

../cont

- Continue this with the details of how to find the MQ queue

```
<service name=”SearchService”>
  <port name=”SearchPort” binding=”tns:SearchFixedBinding”>
    <mq:client QueueManager=”MY_DEF_QM”
               Queuename=”MY_FIRST_Q”
               AccessMode=”send”
               ReplyQueueManager=”MY_DEF_QM”
               ReplyQueueName=”REPLY_Q”
               CorrelationStyle=”messageId copy”
    />
  </port>
</service>
```

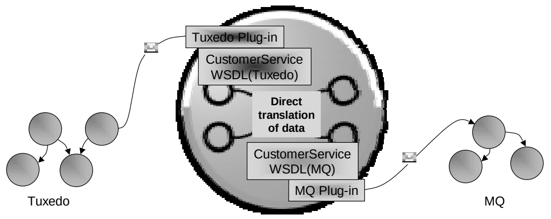
More complex use of MQ is also possible.

“Fixed” binding is the norm because most MQ applications use this. SOAP binding is also supported.

../cont

- Provide our WSDL description to the switch.
  - Now it can understand the interface
  - ..and it can accept incoming Web Services calls and forward them to the underlying MQ service
- Translate the WSDL description into Tuxedo
  - Now the switch can accept incoming Tuxedo calls and forward them to the underlying MQ service

Zoom in on the switch



The use of WSDL means that the core of the switch has to deal with only one way of defining interfaces. Plug-ins allow it to understand WSDL specialised to different middleware (MQ, Tuxedo, CORBA, J2EE, TIBCO/Rendezvous, …), and other plug-ins allow it to send/receive messages using these middlewares.

Case study…

An example of the benefits

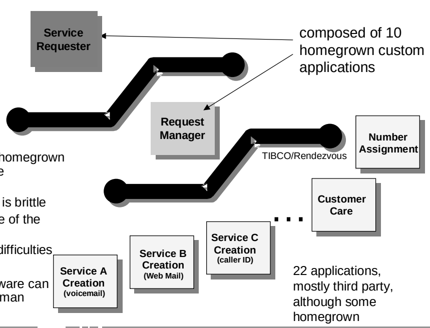
- A Customer Service application for a large Telco
  - involving thirty-two different applications
- 12 of these 32 applications are custom built
  - the others are supplied by third-parties
- Four different types of middleware are employed (actually 12 different versions):
  - MQ, CORBA, Tuxedo, TIBCO/Rendezvous
- Synchronous, Asynchronous, and Publish/Subscribe middleware models in use
- 4 hardware platforms; 3 programming languages

… and all of that in just one business function!

Business case for better integration

- The faster the system can be expanded to provide more services, the faster the company can obtain revenue, gain customers, and make existing customers happy.

System Architecture


Service Requester
composed of 10 homegrown custom applications
Request Manager
Number Assignment
TIBCO/Rendezvous
Customer Care
Service A Creation (voicemail)
Service B Creation (Web Mail)
Service C Creation (caller ID)
. . .
22 applications, mostly third party, although some homegrown

Currently they have a homegrown solution for middleware integration:

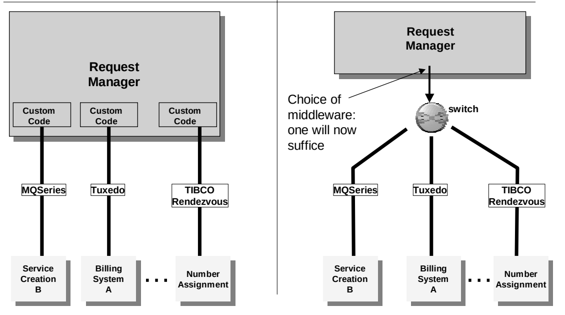
- The environment is brittle
- High cost of some of the middleware
- Support costs & difficulties
- Training costs
- No single middleware can act as the middleman

Original System

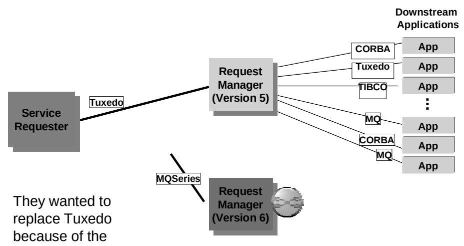
Custom integration per application

Middleware Mediation

Simplify the Request Manager and allow new services to be added easily.

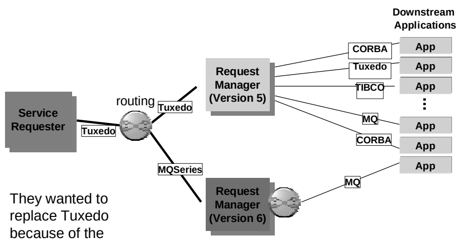


Integrating the new Resource Manager into the system



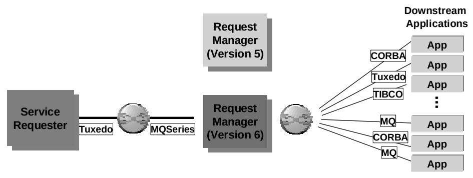
They wanted to replace Tuxedo because of the high cost.

First application switched over



They wanted to replace Tuxedo because of the high cost.

All applications switched over



Tuxedo isn't quite removed, but at least its use is under control.

- To go further, they need to either rewrite the Service Requester (a few applications), or mimic the small part of the Tuxedo API that they are using.

Questions?

More Information

http://www.iona.com/artix
info@iona.com