AN ARCHITECTURAL PATTERN FOR ADAPTABLE MIDDLEWARE INFRASTRUCTURE

University of North Florida
Jason Mitchell

Arturo Sanchez Ph.D.
Project Director
Project Goals

- To define an architectural pattern that allows applications to adapt to new middlewares easily.
  - Separates the operations of the logic from the technology used to facilitate distributed communications
  - Minimal effort to adapt and stabilize.
Problem Domain

- Distributed computing
  - Accomplished with some middleware
  - Must have code in place on both sides to support infrastructure
Common Implementation

- Host → Service via CORBA
- Host → Service via Web Services
- Host → Service via RMI
Disadvantages

- Infrastructure coupled with business logic
  - The business logic is intimately tied to the middleware API.
  - The messages are defined by the interface definition of the language for marshaling purposes.
1st Problem: Direct Access to API

- **RMI**
  - **Client**
    - Obtain reference to rmi stub
    - Pass it around and invoke remote methods on it
  - **Server**
    - Implement business logic in implementation of rmi interface (after all it is an interface)

- **CORBA**
  - **Client**
    - Narrow your reference
    - Pass it around and invoke methods on it
  - **Server**
    - Implement logic directly
2nd Problem: Message Marshaling

- Complex types must be marshaled across the network.
  - RMI
    - Messages are serialized java types
    - Define a message type
      - Now the messaging is tied to the infrastructure
  - CORBA
    - Complex types are defined by the interface definition language.
      - The middleware does the marshaling of the complex types.
Solution

- Apply a behavioral pattern to both sides to decouple the infrastructure
  - Centralize and abstract the API to the middleware from the host application using it.
  - Decouple the message interpretation of complex types from the middleware infrastructure.
Centralize the Infrastructure

- Wrap the middleware and never expose the API
  - This will decouple the technology from either side
- Use an adapter pattern to allow for new technologies
Interpret the complex types in the host application.
- Perform message mapping to internal proprietary types
- Use some message bridge to map the objects into a common form understood by all middlewares.
public Team GetTeamById(int teamId)

// obtain proxy to server object
ITeamService teamService = GetRemoteTeamServiceObject();

GetTeamReq req = new GetTeamReq();
req.teamId = teamId;

// get request string
string request = AppServices.ConvertObjectToString(req);

string response = teamService.GetTeamById(request);

// convert string response into team object
Team team = (Team)AppServices.ConvertStringToObject(response, typeof(Team));

// return object
return team;
[WebMethod]
public string GetTeamById(string request)
{
    // forward the request to the business layer
    return new TeamService().GetTeamById(request);
}
public string GetTeamById(string request) {
    // this is how we convert a string into our
    // application logic request object
    GetTeamReq req = (GetTeamReq) AppServices.ConvertStringToObject(request, typeof(GetTeamReq));

    // now using a team (entity) manager obtain it's info
    TeamMgr mgr = new TeamMgr();
    Team team = mgr.GetTeamById(req.teamId);

    // now serialize the team object back into a string
    string response = AppServices.ConvertObjectToString(team);

    return response;
}
Conclusions

- Only had to map message interpretation once.
- Each new middleware didn’t change any of my application logic.
  - No re-compiling
  - No re-testing
- We were able to adapt to each new middleware easier.
References

  - See http://www.microsoft.com/usa/webcasts/ondemand/960.asp


  - See http://www.dotnetremoting.cc/