JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 2, March-April 2009

WCF: A Case Study Involving a Distributed Client/Server Game

Richard Wiener

1 BACKGROUND

Windows Communication Foundation (WCF) is a software development kit developed by Microsoft as part of .NET 3.0. It provides a flexible and powerful implementation of industry standards defined by IBM, Sun, BEA and Microsoft and defines a service-oriented architecture for communication among machines using various communication protocols.

WCF provides a basis for SOA – service-oriented architecture. This paradigm differs from traditional OOP in which tight encapsulation of data is encouraged and supported by classes. With SOA utilizes loosely coupled services. Each service defines a contract to the entities that consume it. It is generally difficult to implement a single interface across many platforms and languages because of the nature of distributed systems. It is essential to implement the interfaces in a generic manner.

Some of the benefits of SOA messaging include: (1) cross-platform integration in which each platform works with its native data types, (2) asynchronous "fire and forget" communication without the client and server having to wait for each other, (3) Security provided by messages containing a security context involving authentication and authorization.

Prior to WCF, Microsoft supported distributed computation first with Component Object Model (COM) followed by DCOM (Distributed Component Object Model) and later .NET Remoting and XML services. WCF integrates all of these into unified software development platform.

2 THE CASE STUDY

Given the broad capability of WCF one could define dozens of interesting applications, each illustrating different aspects of WCF. These would include web-based services, communication through an intranet and process to process communication within the same computer and many combinations of the above.

Richard Wiener: "WCF: A Case Study Involving a Distributed Client/Server Game", in *Journal of Object Technology*, vol. 8, no. 2, March-April 2009, pp. 71-90
http://www.jot.fm/issues/issue_2009_03/column6/

The case-study presented in this paper focuses on a simple Windows Form client-server intranet distributed game in which communication (messages) originate from the client and are directed at the server which then returns information to the client. A TCP connection will be used to provide the protocol for communication between the client and server computers on the same local area network. The relative simplicity of this application (although as you will see this application is non-trivial) underscores the fact that the goal of this paper is not to demonstrate the breadth and depth of WCF capability (it would take a much more complex and larger enterprise application to achieve this), but to demonstrate how a GUI-based host (server) can support one or more clients (also GUI-based) using SOA in conjunction with OOP and WCF.

The distributed game that will be defined is based on the classic game of Battleship. The paper and pencil game was first produced as a commercial game in the 1930s. In this version, the client computer contains a rectangular grid with many cells. The user selects cells with the mouse (up to 10 at any given time) and then clicks the "Fire" button to hit the server's grid with shells at the exact locations specified on the client computer's grid. The goal is to destroy the ships on the server's grid. A ship is destroyed when all its cells are hit.

The server player starts the game with eight ships. Two giant battleships, two cruisers and four smaller PT boats comprise the armada on the server's grid. The goal is for the client to destroy all eight ships on the server with as few shots as possible. To make things more challenging for the client (the shooter), the person managing the server grid (the eight ships) may move the ships subject to some constraints as the game progresses. It is assumed that the client player cannot see the server player's screen during the game.

Each time the "Fire" button is clicked by the client player, the points selected by the client player must be transmitted to the server computer. Small black colored rectangles are used on the server grid to indicate the location of the shots fired by the client. Whenever a "hit" occurs in one of the server player's ships, an explosion sound is generated on the server computer and a red-colored "X" is painted onto the black rectangle that specifies the location of the shell. This red-colored "X" stays with the ship whenever it is moved. When all the ship's cells have been hit (contain red-colored X's), the ship is destroyed.

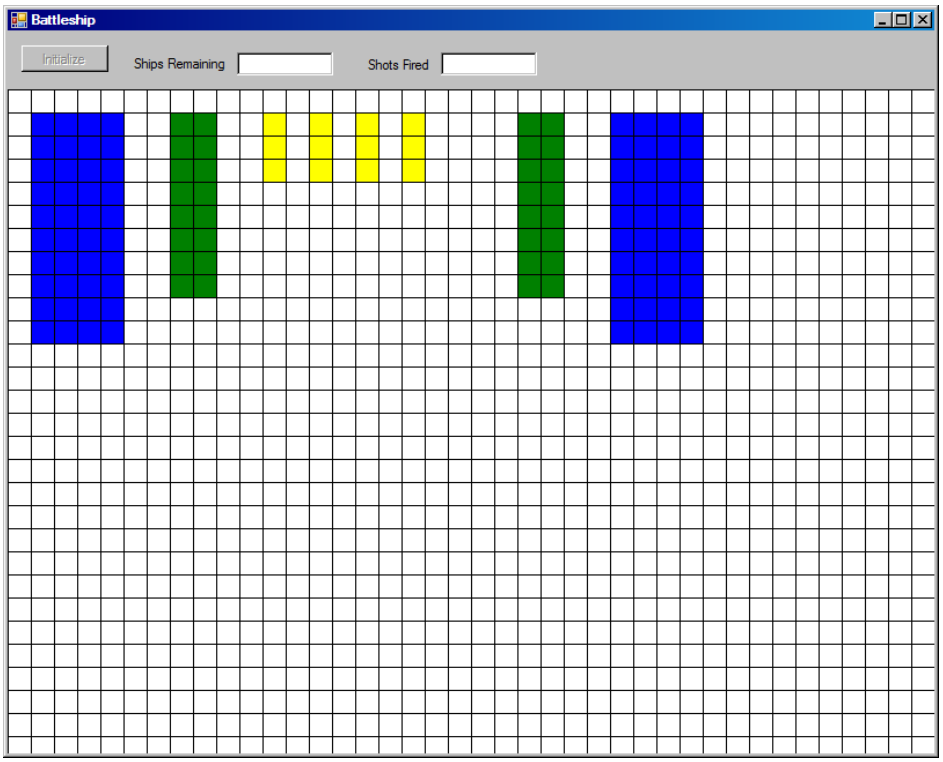
When the client player clicks a cell, a small red rectangle is shown. If the shell hits a server ship the cell remains red otherwise if a miss, the cell turns blue. This provides the client player with visual feedback. Of course the server player can move her ships around during game subject to constraints.

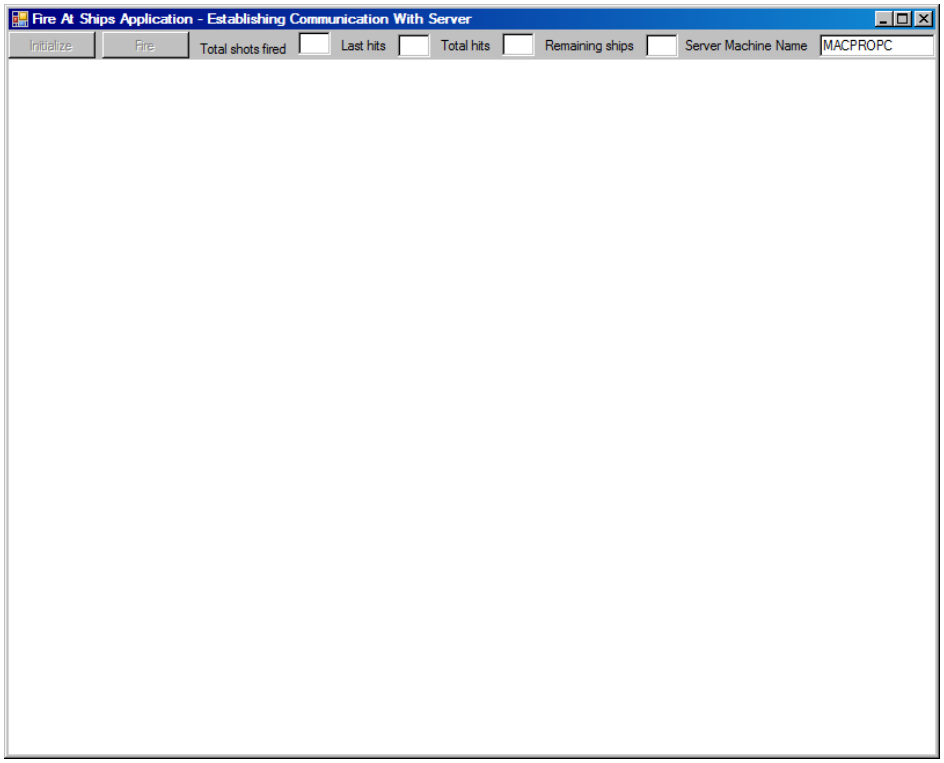
The constraints that govern the movement of server ships are the following:

1. No two server ships can overlap (share any cells).
2. A server ship cannot move past the natural boundaries of the grid.
3. A server ship cannot move if any of its cells in the new position cover a grid location that has been previously fired at. This suggest that it becomes

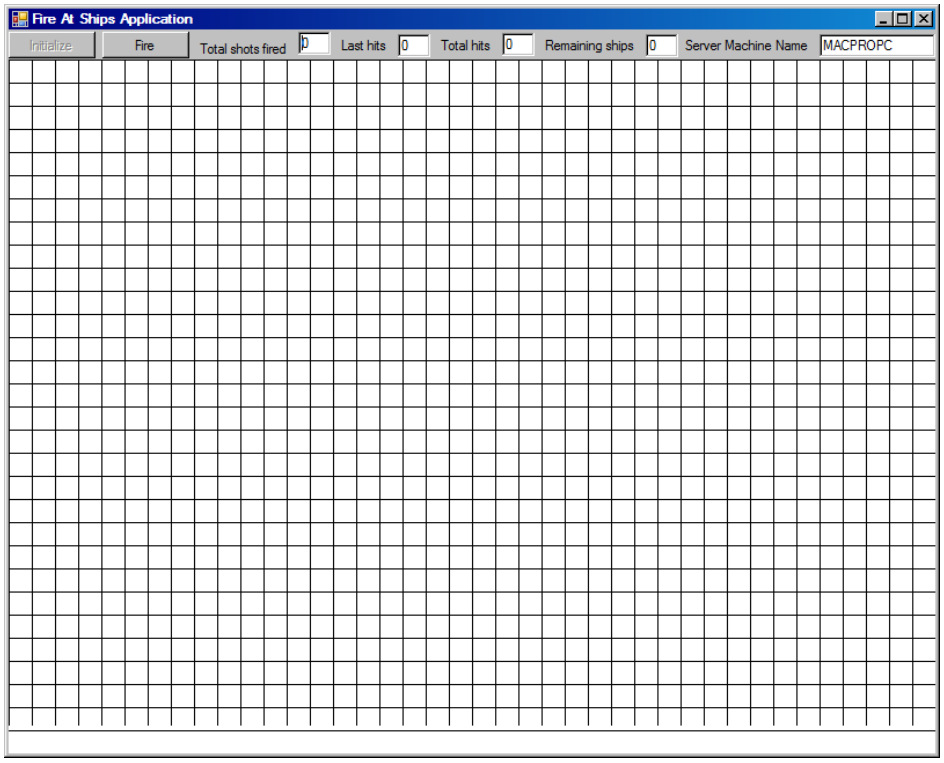
increasingly difficult for the server player to move her ships as the game progresses since a greater number of grid locations will have been fired at.

When the game begins (both the server and client have launched their respective applications), each GUI looks as follows:



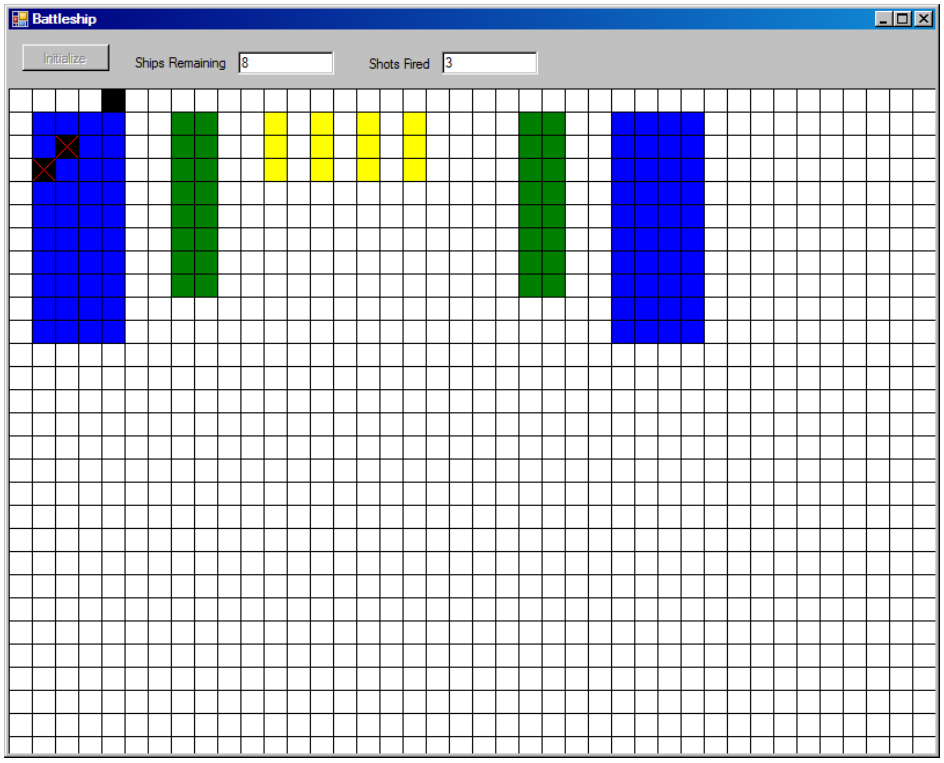


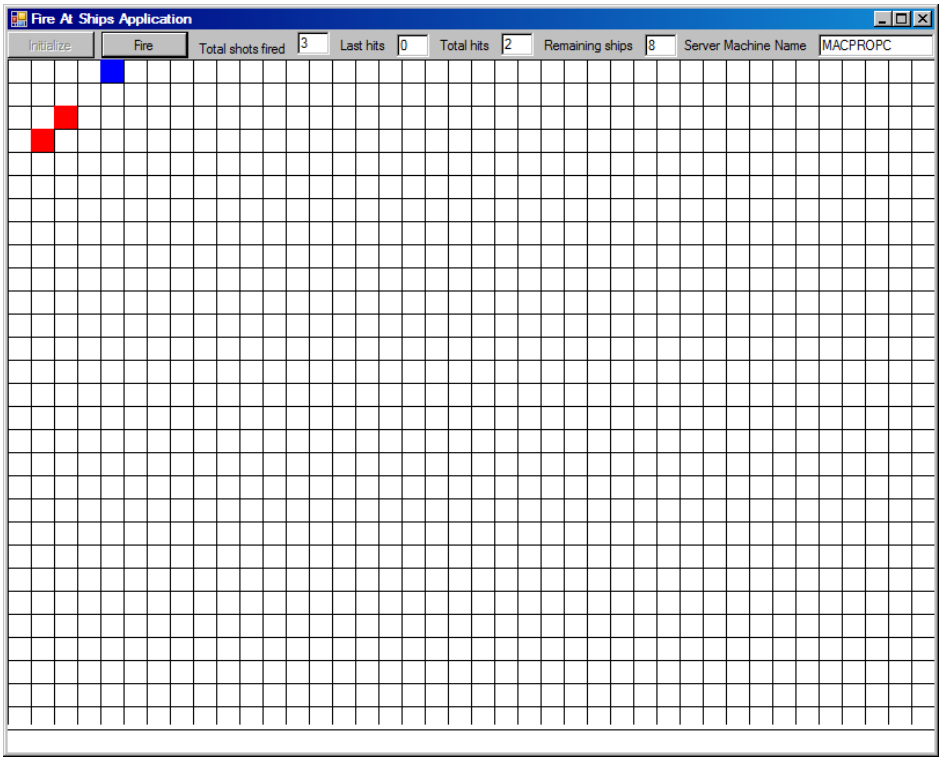
Once the client has established communication with the server (might take 20 seconds), the client application looks like:



The name of the server computer is provided as the right-most field in a *TextBox* control. The server determines its own name and does not need user input for this.

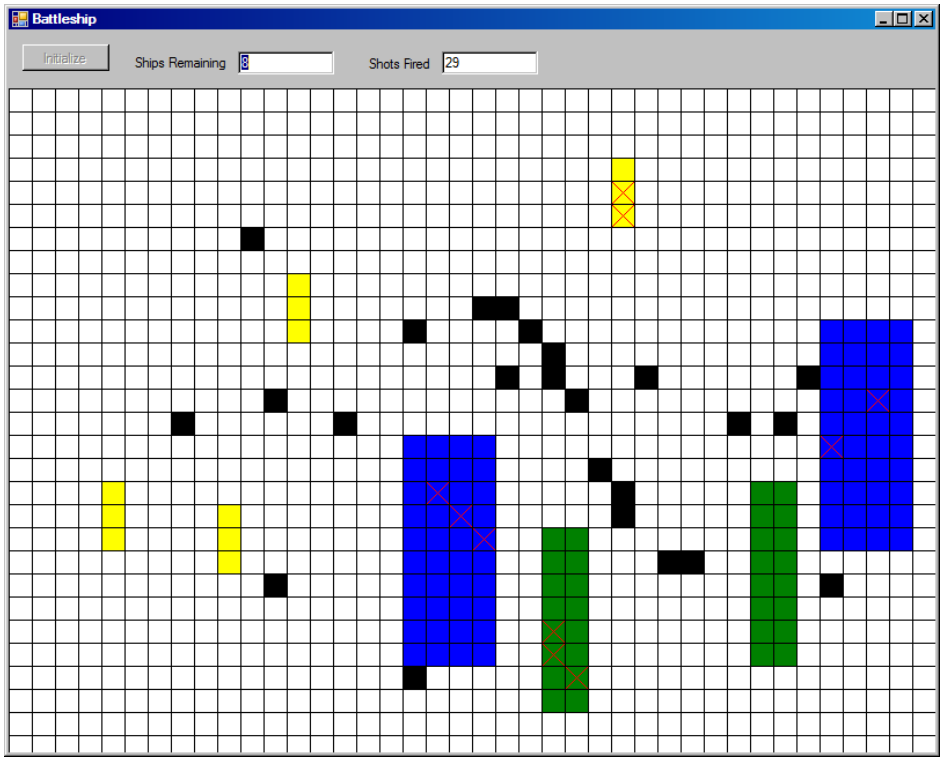
After three shots have been fired by the client, two of which are hits and one miss, the server and client screens look like:

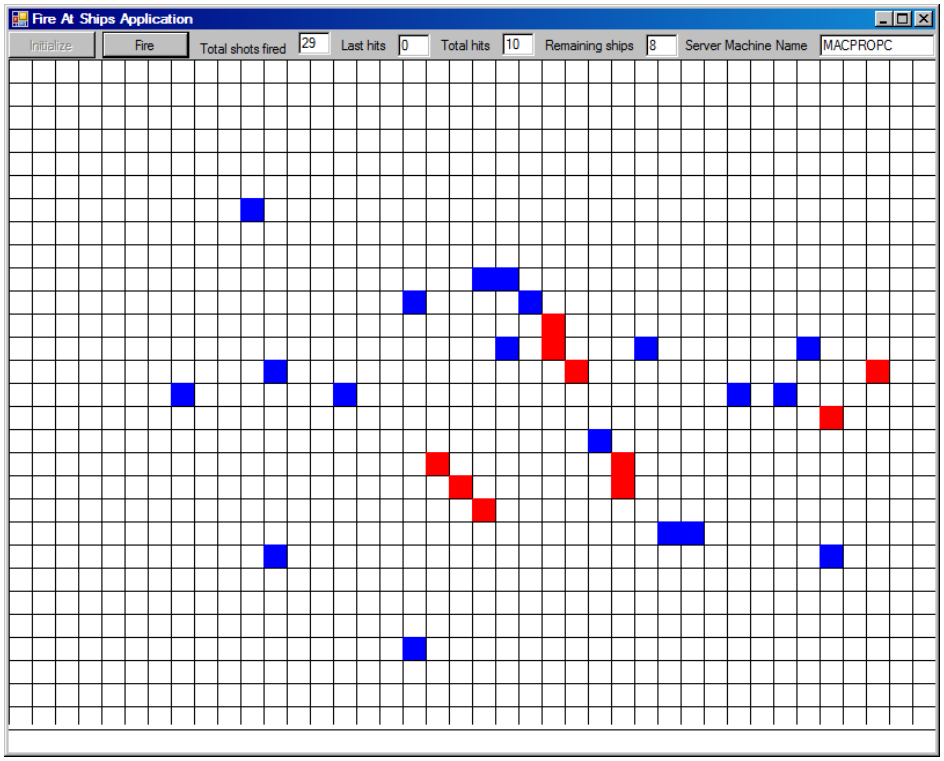




Of course in a real game the server player would have moved her ships from their default initial positions in order to elude the client player.

One final set of screen shots taken partway through a game show typical configurations of the server and client GUI's:





In this game, partially completed, there have been 29 shots fired with only 10 hits scored. These are shown with the red X's on the server grid. The red colored rectangles on the

client grid show where hits occurred. In many cases the hit server ship was moved after the hit occurred.

So playing the game is actually fun for both the client player and the server player. It is also fun building the client and server applications and connecting them using WCF.

An outline of many of the essential elements of the implementation of this client/server game is presented in the next several sections.

3 CONSTRUCTION OF CLIENT AND SERVER APPLICATIONS

We first examine the server host. The code for creating this host is embedded in the usual *Program.cs* file that normally one does not edit or tamper with. The code for this class *Program* follows:

Listing 1 – Class Program that Contains the Service Host

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using System.ServiceModel ;
using System.ServiceModel.Description;

namespace Battleship {
    static class Program {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main() {
            String machineName = Environment.MachineName;
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);

            ServiceHost serviceHost = new
                ServiceHost(typeof(BattleshipUI),
                            new Uri("net.tcp://" + machineName +
                            ":8000/Battleship/BattleshipUI"));

            NetTcpBinding tcpBinding = new NetTcpBinding();


            serviceHost.AddServiceEndpoint(
                                typeof(IBattleshipServices),
                  tcpBinding,   "net.tcp://"   +   machineName   +
                                ":8000/BattleshipUI");

            serviceHost.Open();
            Application.Run(new BattleshipUI());
            serviceHost.Close();
        }
```

```
    }
}
```

The reference **System.ServiceModel.dll** must be added to the Battleship project in order for the WCF bindings to be recognized. The namespaces System.ServiceModel and System.ServiceModel.Description are also used in order to allow unqualified access to the class names *NetTcpBinding* and *ServiceHost*.

The features salient to creating the host programmatically are given by this portion of the code above:

```
ServiceHost serviceHost = new
                ServiceHost(typeof(BattleshipUI),
                            new Uri("net.tcp://" + machineName +
                            ":8000/Battleship/BattleshipUI"));

NetTcpBinding tcpBinding = new NetTcpBinding();


serviceHost.AddServiceEndpoint(
   typeof(IBattleshipServices),
   tcpBinding, "net.tcp://" + machineName +
   ":8000/BattleshipUI");

serviceHost.Open();
```

The class name *BattleshipUI* (the GUI class) is specified in the *ServiceHost* constructor. The “net.tcp” binding followed by the machine name and port 8000 is used as part of the endpoint specification in the *ServiceHost* constructor. The machineName string is obtained from the static property *MachineName* in class *Environment*. The method *AddServiceHost* is invoked on the *serviceHost* object using the typeof operator on the *IBattleshipServices* interface. This interface specifies the service contract that the client application utilizes as well as the callback contract that the server application uses to send information back to the client.

Listing 2 presents the details of the *IBattleshipServices* interface.

Listing 2 – Interface IBattleshipServices

```
namespace Battleship {

    [ServiceContract(SessionMode=SessionMode.Required,
            CallbackContract=typeof(ICallbackServices))]
    public interface IBattleshipServices {
        [OperationContract(IsOneWay = true)]
        void ShootAt(Point pt);
    }
```

```
    public interface ICallbackServices {
        [OperationContract(IsOneWay = true)]
        void Results(int remainingShips, int shotsFired,
                     int totalHits, int lastHits);
        [OperationContract(IsOneWay = true)]
        void ScoreHit(Point pt);
        [OperationContract(IsOneWay = true)]
        void GameOver();
    }
}
```

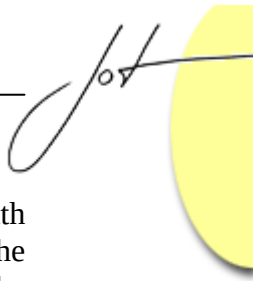
Numerous attributes (meta-data information embedded between rectangular brackets) decorate the interface. Only a single operation service contract, *ShootAt*, is specified. A point object must be passed from the client to the server (the location where the shell explodes).

Three operation service contracts are specified as callback message signatures in which the server can inform the client application about the number of remaining ships, total number of shots fired, total number of hits and last number of hits (on the last firing by the client against the server). That is all taken care of in the operation callback signature for Results. The *ScoreHit* callback operation contract provides the basis for communicating the location of a hit to the client. Finally, the *GameOver* callback operation contract allows the server to inform the client that the game is over because all ships have been destroyed.

On the other end of the system, Listing 3 shows how the client application connects to the server application. Listing 3 presents the code activated by the client player clicking the *InitializeBtn* button.

Listing 3 – Button handler for the Initialize button

```
private void InitializeBtn_Click(object sender, EventArgs e) {
   InitializeBtn.Enabled = false;
   serverMachineName = serverMachineNameTextBox.Text;
   EndpointAddress ep = new
   EndpointAddress("net.tcp://" + serverMachineName +
         ":8000/Battleship/BattleshipUI");
    this.Text = "Fire At Ships Application - Establishing
                Communication With Server";
    DuplexChannelFactory<IBattleshipServices> factory =
         new DuplexChannelFactory<IBattleshipServices>(
               new FireAtShipsUI(), new NetTcpBinding(), ep);
    // This allows a client computer with different user name and
    // password to login to the server/
    // The USER_NAME_STRING and PASSWORD_STRING
    // must be replaced with actual string names
    factory.Credentials.Windows.ClientCredential =
         new NetworkCredential(USER_NAME_STRING, PASSWORD_STRING);
    proxy = factory.CreateChannel();
    proxy.ShootAt(new Point(-1, -1));
    // Other code not related to the WCF communication
}
```



A “blank” shot at Point(-1, 1) is done to cement the communication along the path provided. A *DuplexChannelFactory* is used to establish two-way communication. The endpoint address requires the name of the server computer. This is given as part of the user interface (a *TextBox*) in the *FireAtShipsUI* class. The *DuplexChannelFactory* constructor specifies a base type of *IBattleshipServices*, the communication protocol (*NetTcpBinding*) and the endpoint address of the server. When the client user clicks the *InitializeBtn* button it is assumed that the server host has been activated.

Listing 4 contains all the details of the important client GUI class *FireAtShipsUI*.

Listing 4 – Class FireAtShipsUI

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.ServiceModel.Channels;
using Battleship;
using System.Threading;
using System.Net;

namespace FireAtShip {

    [ServiceContract(SessionMode = SessionMode.Required,
            CallbackContract = typeof(ICallbackServices))]
    public interface IBattleshipServices {
        [OperationContract(IsOneWay = true)]
        void ShootAt(Point pt);
    }

    [ServiceContract(SessionMode = SessionMode.Required,
            CallbackContract = typeof(ICallbackServices))]
    public interface ICallbackServices {
        [OperationContract(IsOneWay = true)]
        void Results(int remainingShips,
           int shotsFired, int totalHits, int lastHits);
        [OperationContract(IsOneWay = true)]
        void ScoreHit(Point pt);
        [OperationContract(IsOneWay = true)]
        void GameOver();
    }

    public delegate void LabelUpdate();
    public delegate void TextUpdate(TextBox textbox,
                                    String someValue);

    [CallbackBehavior(UseSynchronizationContext = false)]
    public partial class FireAtShipsUI : Form,
```

```
ICallbackServices {

    // Fields
    private Graphics g;
    private IBattleshipServices proxy;
    private List<Point> fireAt = new List<Point>();
    private Thread timer;
    private int numberShots;
    private String serverMachineName;

    public FireAtShipsUI() {
        InitializeComponent();
        g = this.panel.CreateGraphics();
        FireBtn.Enabled = false;
        gameOverLbl.Visible = false;
    }

    private void UpdateStatus() {
        while (true) {
            try {
                Thread.Sleep(1000);
                AssignToTextBox(totalHitsTextBox, "" +
                                Global.totalHits);
                AssignToTextBox(remainingShipsTextBox, "" +
                                Global.remainingShips);
                AssignToTextBox(lastHitsTextBox, "" +
                                Global.hits);
                AssignToTextBox(totalShotsFiredTextBox, "" +
                                Global.shotsFired);
                foreach (Point pt in Global.hitsScored) {
                    g.FillRectangle(new SolidBrush(Color.Red),
                                pt.X, pt.Y, 20, 20);
                }
                if (Global.GAME_OVER) {
                    MakeLabelVisible();
                }
            } catch (Exception) { }
        }
    }

    private void MakeLabelVisible() {
        if (!this.InvokeRequired) {
            gameOverLbl.Visible = true;
            FireBtn.Enabled = false;
        } else {
            Object[] parameters = { };
            this.Invoke(new LabelUpdate(MakeLabelVisible),
                        parameters);
        }
    }

    private void AssignToTextBox(TextBox textbox,
                        String someValue) {
        if (!this.InvokeRequired) {
            textbox.Text = someValue;
```

```
        } else {
            Object[] parameters = { textbox, someValue };
            this.Invoke(new TextUpdate(AssignToTextBox),
                                    parameters);
        }
    }


    public void Results(int remainingShips, int shotsFired,
                            int totalHits, int lastHits) {
        Global.totalHits = totalHits;
        Global.shotsFired = shotsFired;
        Global.hits = lastHits;
        Global.remainingShips = remainingShips;
    }

    public void GameOver() {
        Global.GAME_OVER = true;
    }

    public void ScoreHit(Point pt) {
        Global.hitsScored.Add(pt);
    }

    private void InitializeGrid() {
        for (int x = 0; x < 40; x++) {
            g.DrawLine(new Pen(Color.Black, 1),
                            x * 20, 0, x * 20, 575);
        }
        for (int y = 0; y < 30; y++) {
            g.DrawLine(new Pen(Color.Black, 1), 0,
                        y * 20, 800, y * 20);
        }
    }

    private void InitializeBtn_Click(object sender,
                            EventArgs e) {
        // Code presented in Listing 3
        this.Text = "Fire At Ships Application";
        InitializeGrid();
        timer = new Thread(new ThreadStart(UpdateStatus));
        timer.IsBackground = true;
        timer.Start();
        FireBtn.Enabled = true;
    }

    private void FireBtn_Click(object sender, EventArgs e) {
        lock (fireAt) {
            foreach (Point pt in fireAt) {
                // Communication with server
                proxy.ShootAt(pt);
                g.FillRectangle(new SolidBrush(Color.Blue),
                                    pt.X, pt.Y, 20, 20);
            }
            fireAt.Clear();
            numberShots = 0;
```

```
            }
        }

        private void panel_MouseDown(object sender,
                                     MouseEventArgs e) {
            if (numberShots < 10) {
                int x = e.X / 20 * 20;
                int y = e.Y / 20 * 20;
                g.FillRectangle(new SolidBrush(Color.Red),
                                x, y, 20, 20);
                lock (fireAt) {
                    fireAt.Add(new Point(x, y));
                }
                numberShots++;
            }
        }
    }
}
```

The first observation is that the interface for *IBattleshipServices* must be included in the client application (here it is done inline in the file that contains class *FireAtShipsUI*.

An *UpdateStatus* thread is launched (see the UpdateStatus method above). Every second it grabs static information contained in a class *Global* and uses this information to update text fields in the class. In order to avoid cross-thread updates of the GUI (from thread *UpdateStatus* to the main GUI thread), two delegates (*LabelUpdate* and *TextUpdate*) are defined. So, for example, to update the textbox with someValue, first the else clause in the if-else construct will be executed. This in turn will cause a recursive invocation of the AssignToTextBox method which will directly update the desired text box. But it will do so from within the main GUI thread thus avoiding the dreaded cross-thread exception or cross-thread flaky behavior from occurring. The same strategy is used when the label that announces the end of the game is made visible.

When the server informs the client through the ScoreHit method that a hit has occurred on the client grid, this information is immediately passed to the static *hits* List contained in class *Global*.

Listing 5 presents the code for class Global.

Listing 5 – Class Global

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Battleship;
using System.Drawing;
using System.ServiceModel;

namespace FireAtShip {

    public class Global {
        public static int totalHits;
        public static int hits;
```

```
        public static int remainingShips;
        public static int shotsFired;
        public static List<Point> hitsScored =
                    new List<Point>();
        public static bool GAME_OVER;
    }
}
```

The server application is more complex than the client. It needs to provide the player the ability to move ships with the mouse within the constraints stated earlier. It needs to determine whether a shell hits or misses a ship and then must inform the client of the outcome.

Classes *BattleshipUI* (the server GUI class), *Assets* (containing the business model behind the GUI class) and *Ship* (another business class that fires events) from the basis of the server application.

Listing 6 presents the two delegate types that are used to provide event firing and communication between the business model and listener (GUI class) class.

Listing 6 – Delegate Classes to support Server Application

```
namespace Battleship {

    public delegate void EraseOldShipAction(Point pt,
                            int width, int height);

    public delegate void DrawNewShipAction(Point pt,
                int width, int height, Color color, List<Point> hits);

}
```

Several important methods from class Ship are shown in Listing 7.

Listing 7 – Several important methods from class Ship

```
 public void MoveShip(Point newPt) {
   Point oldPt = location;
   // Erase old ship
   FireEraseShape(oldPt, width, height);

   this.Location = newPt;
   List<Point> movedHits = new List<Point>();
   foreach (Point pt in Hits) {
   movedHits.Add(new Point(pt.X + newPt.X - oldPt.X,
                           pt.Y + newPt.Y - oldPt.Y));
   }
   hits.Clear();
   foreach (Point p in movedHits) {
   hits.Add(p);
```

```
    }
    FireDrawShape(newPt, width, height, color, Hits);
    hasMoved = true;
}

public void FireEraseShape(Point oldPt, int width,
                           int height) {
    if (erase != null) {
    erase(oldPt, width, height);
    }
}

public void FireDrawShape(Point newPt, int width,
                     int height, Color color, List<Point> hits) {
    if (draw != null) {
        draw(newPt, width, height, color, hits);
    }
}
```

The events *draw* and *erase* are fired within the methods *FireEraseShape* and *FireDrawShape* respectively. These events are registered with listener methods within the *BattleshipUI* GUI class.

Two important and interesting methods within class Assets are presented.

Listing 8 – Two important methods from class Assets

```
public bool NoCollisionWithOtherShipsAndBoundary(
                Ship selectedShip, int x, int y) {
    // Iterate over points in selectedShip
    int rows = selectedShip.Height / 20;
    int cols = selectedShip.Width / 20;
    for (int xPos = x; xPos < x + cols * 20; xPos += 20) {
        for (int yPos = y; yPos < y + rows * 20; yPos += 20) {
            if (PointInShip(new Point(xPos, yPos)) != null &&
               PointInShip(new Point(xPos, yPos)) != selectedShip ||
               xPos < 1 || xPos > 39 * 20 || yPos <= 0 ||
               yPos >= 29 * 20 ||
               Global.fireAt.Contains(new Point(xPos, yPos))) {
                   return false;
             }
         }
     }
     return true;
 }

public Ship PointInShip(Point pt) {
    for (int index = 0; index < 8; index++) {
        if (ships[index] != null) { // ship has not been destroyed
            int shipX = ships[index].Location.X;
            int shipY = ships[index].Location.Y;
            int shipWidth = ships[index].Width;
            int shipHeight = ships[index].Height;
```

```
            if (pt.X >= shipX && pt.X < shipX + shipWidth &&
                pt.Y >= shipY && pt.Y < shipY + shipHeight) {
                return ships[index];
            }
        }
    }
    return null; // pt not contained within any ship
}
```

Listing 9 presents the *UpdateStatus* thread that is launched within the *BattleshipUI* class. Like its counterpart in the client application, it updates the server GUI every second and sends important information back to the client.

Listing 9 – UpdateStatus Thread

```
private void UpdateStatus() {

   int hits = 0;
   List<Point> scoreHits = new List<Point>();
   while (true) {
         canMove = true;
   Thread.Sleep(2000);
         canMove = false;
      ICallbackServices callback = null;
      if (Global.fireAt.Count > 0 && !mouseDown) {
         lock (Global.fireAt) {
            hits = 0;
            foreach (Point pt in Global.fireAt) {
               Ship ship = assets.PointInShip(pt);
               if (ship == null) {
               DrawShape(pt, 20, 20, Color.Black, null);
               }
               if (ship != null && !ship.Hits.Contains(pt)) {
                   scoreHits.Add(pt);
                   callback =
 Global.context.GetCallbackChannel<ICallbackServices>();
                   callback.ScoreHit(pt);
                   hits++;
                   totalHits++;
                   ship.Hits.Add(pt);
                   ship.FireDrawShape(pt, 20, 20,
                                Color.Black, ship.Hits);
                   // Executes explosion
                   player.Play();
                   Thread.Sleep(2000);
                   if (ship.Hits.Count == ship.HitCapacity) {
                     shipsRemaining--;
                     if (shipsRemaining == 0) {
                     callback.GameOver();
                     }
                   }
                   callback.Results(shipsRemaining,
                           Global.shotsFired, totalHits, hits);
```

```
                } else {
                callback =
Global.context.GetCallbackChannel<ICallbackServices>();
                   callback.Results(shipsRemaining,
                            Global.shotsFired, totalHits, hits);
                }
            }
          // Additional code that updates text boxes
```

Several invocations of callbacks to the client are evident in the *UpdateStatus* thread.

Some mouse handling event code that allows the server player to move a ship is presented in Listing 10.

Listing 10 – Mouse handling code to move ships

```
private void panel_MouseDown(object sender, MouseEventArgs e) {
   mouseDown = true;
   oldPt = new Point(e.X, e.Y);
}

private void panel_MouseUp(object sender, MouseEventArgs e) {
   if (canMove) {
         Ship selectedShip = assets.PointInShip(oldPt);
         int x = e.X / 20 * 20;
         int y = e.Y / 20 * 20;
         if (selectedShip != null &&
               assets.NoCollisionWithOtherShipsAndBoundary(
                                 selectedShip, x, y)) {
               selectedShip.MoveShip(new Point(x, y));
               DisplayHits();
         }
         mouseDown = false;
    }
}

private void DisplayHits() {
    lock (Global.fireAt) {
        if (assets != null) {
               InitializeGrid();
               foreach (Point pt in Global.fireAt) {
               g.FillRectangle(new SolidBrush(Color.Black),
                                 pt.X, pt.Y, 20, 20);
            }
            assets.DisplayShips();
         }
     }
}
```

Class Global that contains much of the important status information that is updated every second is given in Listing 11.

Listing 11 – Class Global

```
namespace Battleship {

   public class Global {
        public static List<Point> fireAt = new List<Point>();
        public static OperationContext context;
        public static int shotsFired;
    }
}
```

4 CONCLUSIONS

Although some of the coding details have been omitted because of space constraints and because they are not interesting, it should be evident that WCF provides an excellent basis for client -> server -> client communication governed by service contracts and client callback contracts. These integrate nicely into a “normal” Winform application structure as shown above. The most challenging aspects of the development process for the client and server applications was getting the client application to handshake with and see the server application and later getting the server application to successfully talk back to the client application. This took many attempts and required consulting several WCF books and web postings (this is the first serious WCF application written by the author). It was important to use the

```
if (!this.InvokeRequired) {
} else {
}
```

mechanism (used in both the client and server applications) for updating GUI controls since in both cases a UpdateStatus thread was used to trigger the GUI control updates. Without using this programming pattern either unexpected behavior would occur or no updates would be posted to the GUI controls. It was critically important to use the DuplexChannelFactory class in the client to facilitate two-way communication.

Another important concern was ensuring that the global data was appropriately locked when accessing or modifying its contents. It was important to block the server player from moving while hits were being recorded in the *UpdateStatus* thread on the server. This is accomplished using the field *canMove* in class BattleshipUI.

Hopefully some of the details of the distributed computing game presented above will be helpful to those learning WCF.

About the author

Richard Wiener is Chair of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 22 books and works actively as a consultant and software contractor whenever the possibility arises. His latest book, published by Thomson, Course Technology in April 2006, is entitled *Modern Software Development Using C#/.NET*.