Implementing Blaise 5 in a production environment

Paul Segel, Mangal Subramanian, Ray Snowden, Richard Frey, Mike Florczyk
Westat, Inc.

Blaise 5 is a fully re-engineered version of the Blaise product with new features and capabilities, a completely rewritten code base in Microsoft .Net, and a new deployment model and management tools. As such, one of the necessary steps for organizations preparing to adopt Blaise 5 is to begin to understand how to deploy, manage, and use Blaise 5 in a production environment particularly where the mechanisms and options in Blaise 5 are different from earlier versions of Blaise.

This paper discusses three aspects to the use of Blaise 5 in a production environment: 1) Deploying Blaise 5 in a server environment; 2) demonstrating how the Blaise 5 API can be used to support selected case management functions; and 3) considerations for conducting stress testing with Blaise 5.

This paper references our experiences working with Blaise 5.0.1.553. Development of the Blaise 5 product is ongoing and there will likely be changes to the software over the next several months. Any bugs or other similar issues are not addressed here because it is assumed they would be fixed in future releases.

1. Deploying Blaise 5

Blaise is a powerful and flexible system used for computer-assisted survey processing. With the introduction of Blaise 5 surveys can be deployed across multiple platforms including web/application servers, mobile devices (smart phones and tablets), desktops, and laptops. The surveys can be deployed as browser-based web-applications, native mobile apps, and as standalone applications. This paper focuses on the deployment of Blaise 5 in a server environment to support access via web browsers or rendered with the Blaise native mobile app.

The primary administrative tool used to deploy and manage Blaise 5 surveys is the Blaise Survey Manager. The Survey Manager is a browser-based tool that supports the following administrative functions:

- Configure and manage server parks
- Deploy surveys to server parks
- Create and manage end user accounts and permissions

1.1. Server parks

Blaise 5 has re-defined the packaging and deployment of the Blaise run time components in order to provide greater scalability, improved system continuity, and enhanced security. Blaise 5 distinguishes several types of logical server roles in the execution of a Blaise instrument. The server roles include:

- Web server – the server with which the end-user directly connects; process requests and send responses.
- Resource server – applies layout and formats the Blaise survey for different devices/modes.
- Data Entry server – handles pages and interprets/applies the Blaise rules.

- Data server– reads and writes data to a Blaise data source.

The collection of servers on which these server roles are installed to support a single Blaise 5 installation is called a server park. One or several server roles may be installed on a single physical or virtual server. Alternatively, for high volume applications and/or to provide redundancy to protect against component failure, any of the server roles, with the exception of the data server role, can be installed as a cluster of 2 or more servers.

The Blaise software is installed on any physical/virtual server which will be used to support a Blaise server role in a Server Park. The Survey Manager provides functions to assign and configure server roles to physical/virtual servers and to manage the distribution at run-time of Blaise-related processing to the server roles across the Server Park.

Under Survey Manager, the ip address and port number over which Blaise 5 communicates is specified for each server role defined. When installing Blaise 5, it is important to work with the network administrators and systems security staff within your organization to make sure that the ports selected for use are consistent with your organization's policies and that the necessary network access has been provided, e.g., through firewall settings.

Blaise 5 server roles can be installed either on physical or virtual servers. In our test installation, we used a Server Park with one virtual server hosted on VMWare running Windows 2008 R2, service pack 1 and used SQL Server 2008 on a separate virtual server for database support. The use of virtualization to support Blaise 5 provides additional flexibility and efficiencies over a physical server installation including:

- New servers can be quickly deployed on existing hardware.
- Additional resources (CPU, memory, disk space) can quickly be added to an existing server with little disruption.
- Virtual servers can be moved from one virtual host (physical server) to another very quickly in the event of hardware problems, maintenance, etc.
- Copies/clones of virtual servers can be created and saved for system recovery.

Blaise 5 no longer supports the use of the proprietary BDB file to store data but rather uses a DBMS for database services, such as SQL Server, or the default SQLite which is installed with Blaise 5. In addition to the Blaise Server Park described above, a Blaise 5 installation must also include a server running a supported DBMS.

1.2. Deploying surveys

When a survey is ready for deployment a survey installation package is created. The package is created in the Blaise Control Centre where source code is converted into executable code and other survey files are packaged together. The package contains the questionnaire's binaries and supporting files, together with layout and runtime settings.

The Survey Manager is used to install and manage survey packages on Server Parks providing such functions as activation, deactivation, previewing and removing surveys. You can also view details about a survey such as the data source, the active status of the survey, and the installation date. All these functions are handled by an administrator.

In software development, it is common practice to deploy an application through a series of environments including development, testing, acceptance and production. In Blaise 4, moving a Blaise instrument to different environments required manual reconfiguration of the Blaise OLEDB Interface (boi) files to specify the appropriate database. In Blaise 5 the Control Centre allows you to create different Blaise Data Interface (bdix) files, and assign a role to each file representing development, testing, acceptance or production. When the corresponding role is set in the creation of a deployment package through Control Center, the bdix file with the matching role will be used.

The Blaise 5 Server Manager also supports the deployment of a survey package to different testing environments. Server Parks can be used to support the different environments and make the survey available to the appropriate testers whether they are testing on a Windows desktop, a browser, a mobile device or a tablet. The bdix files can be used as described above to help allow packages to be easily deployed to different environments.

1.3 User Management

The Server Manager provides functions to create user accounts that are used to authenticate access to Survey Manager and assign permissions to access various functions. A built-in account called Root is the default Blaise 5 administrator account and is the only account that can create and manage users.

With the User Management function you can control which server a user can access and what type of changes these users can make to the servers and the installed surveys. Each person can have a separate user account with unique settings and preferences. The settings can be as detailed as allowing a user access to just one survey or as broad as allowing a user control over all server parks.

2. Security

Blaise 5 production instruments will need to comply with an organization's security policies. Rhoads and Snowden in their 2010 IBUC paper quote the FISMA definition of information security as:

> Protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide the following general safeguards:
>
> • Confidentiality – restricting access to information to authorized users only. This is a central security concern in survey research projects where respondent data often includes highly confidential personal information and PII. Unauthorized disclosure of confidential data is everybody's top concern.
>
> • Integrity - guarding against the unauthorized modification or destruction of information due to either accidental or malicious actions. This is another important aspect of security in survey research projects where survey data is generally time-consuming and expensive to collect and the credibility of analytic findings depends on a high degree of confidence in the quality of the underlying survey data.
>
> • Availability - ensuring timely and reliable access to and use of applications and information. Consistent and reliable access to survey systems in CAI projects can be particularly critical since response rates are extremely important. Once a respondent is contacted and ready to provide information, the CAI systems must work.

We will consider various aspects of the Blaise 5 system with respect to those safeguards.

2.1. Servers

We installed Blaise 5 in our secure data center on virtualized Windows 2008 R2, SP1 servers created using a secure server configuration template, regularly scanned for security vulnerabilities, and maintained with current security patches. Virtual images, backed up regularly, are maintained on dual SAN storage to provide redundancy for disaster recovery. Database servers are maintained in a separate security zone with network isolation provided by a firewall. This isolates database servers from direct access by end-users over the Internet. Blaise 5 has been compatible with our standard security templates, which maintain secure policies. We will discuss sizing and performance measurement in a later section of this paper.

2.2. Server Parks and firewalls

Blaise 5 Server Parks provide a ready context for implementing network isolation, continuity, and recovery. To scale out capacity, multiple servers may be employed to prevent overloading the Web and Data Entry servers (running the DataEntry and Resource services), with load balancing provided by the Management server, and accessing data through a single Data Server. Actual production databases, accessed through the Data server may be isolated by a firewall with normal port access for the database.

The port to the Management server is configurable when the server park is deployed. The Management server supports both internal and external functions. Internally it supports the configuration of server roles, the deployment of instruments, and user management. Externally, it can be used to list available instruments to a user. At this time, the Blaise team is considering supporting those functions over different ports allow tighter security for external users.

2.3. Connected data collection

Blaise 5, especially with its support for mobile devices, offers new models for configuring the relationship between central servers and remote data collection devices. These devices include remote web browsers, laptops, tablets, and mobile phones. Many of the issues associated with remote data collection devices are familiar and have been part of previous Blaise versions. We will try to describe some issues related to Blaise 5 specifically and the new devices that it supports.

Blaise 5 can be installed and operated to collect data in two different configurations - connected data collection and disconnected data collection. The two configurations share a number of common security considerations. With its tighter connection to the server, we will start with connected data collection.

In a connected configuration a number of security issues are handled by the central servers including:

- Authentication and authorization – these gateway functions might be performed external to the Blaise 5 instrument as part of the mechanism associated with launching the instrument or accessing the remote device. Within a Blaise 5 instrument, text can be hidden to shield password entry, and the proposed alien procedure could be used to encrypt and compare passwords. A second authentication factor is still probably best handled outside the Blaise 5 instrument

- Security of the configuration information - Blaise 5 encrypts sensitive elements of the configuration such as any connection strings in the data interface file (.bdix). Further the Control Centre can enforce password protection on its decryption.

- Security of data during collection – in the connected collection mode, data responses are transmitted from the device to the central servers. For web surveys, SSL encryption can be enforced to Blaise 5 web sites.

- Security of data collection at rest – with the data residing on central servers, the security of the data becomes an issue of access control to the server park servers and any attached database servers and possibly data encryption.

Security considerations for the client device in a connected configuration include:

- Application and security configuration on the device – organizations need to configure the remote device to comply with any required security and operational standards. As Blaise 5 nears release we will be testing compatibility with security standards such as the NIST United States Government Configuration Baseline (USGCB) for applicable operating systems, and with accessibility standards, such as Section 508.

- Application, security, and operating system updates - although data are not stored on the remote device, it is a gateway to the central servers. It is important to maintain the security features of the device with current patches.

  With Blaise 5, mobile devices, such as iOS and Android phones and tablets, are running native applications to interface with the central servers. Updates to these applications need to be evaluated for compatibility, risk, and benefit. Procedures for version upgrades and enforcement will be required.

- Similarly, an organization's applications, possibly from a private application store, such as through the iOS Developer Enterprise Program, and instruments need to be maintained and configured in a secure manner.

- Loss or damage to the device – Blaise 5 introduces new device types, but otherwise an organization needs to maintain its current procedures for reporting and addressing the loss or damage to the remote device.

2.4. Disconnected data collection

Data collection on disconnected devices adds additional considerations and new complications to the issues associated with connected devices. These include:

- Protecting data at rest – strategies that have been developed for platforms used with earlier Blaise versions, will need to be re-evaluated with Blaise 5 and extended to new devices and storage platforms. For example, methods for maintaining encrypted data may need to migrate to new storage and device platforms.

- Protecting data transmission – similar re-tooling may be required for maintaining secure transmission as data collected on the remote device is transmitted to central servers, as is normally required.

- Loss or damage to the device – with data stored on the remote device, even in a secure form, it may be more important to consider tracking and data wiping capabilities for disconnected remote devices.

3. Some case management functions with API

The Blaise 5 API can provide the tools to perform some case management functions. We will discuss an environment where the Blaise 5 API provides some functions in a component (it could be transformed into a Web Service) that can be used by a case management application (or web site). For purposes illustration we will assume that external data resides in SQL Server, and we will deal with database tables as the primary method of sending and receiving external data.

The sample component, BlaiseAgent, provides a wrapper around the Blaise 5 API, and creates functions for some common data management functions. These case management functions might include:

- Preloading data for a case
- Returning case-level status information
- Extracting case results

The Blaise 5 API is divided into four components, which current documentation describes as:

- Meta API for read-only access to metadata information
- DataRecord API for data validation, checking and routing
- DataLink API for reading/writing of stored data
- SessionData API for read-only access to session data

Because of the nature of our sample tasks, our sample component will most heavily focus on the Meta, DataRecord, and DataLink APIs . In discussing the code, we will frequently qualify an object with the name of the API to which it belongs.

The BlaiseAgent component may return Blaise 5 objects ,so a calling application may require references to Blaise 5 components including Statneth.Blaise.Data.SQLite and possibly any other Blaise 5 API that contains the returned object.In this discussion, the BlaiseAgent component contains a single class Agent. Agent is initialized with the file name of the data model (bmix) and the database (bdix) and owns the corresponding objects:

- MetaAPI Datamodel object named: dm
- DataLinkAPI IDataLink object named: dl

We will look at several public methods the Agent exposes. For these examples we are ignoring block structures in the data model. (They would be handled by recursing through the fields in the block.)

3.1. Preload data

This example code writes records to a Blaise data base from a SQL Server data table.

```
1  public void Load(DataTable dt)
2    {
3     ICollection<DataRecordAPI.IDataRecord> cDR = new   List<DataRecordAPI.IDataRecord>();
4     foreach (DataRow r in dt.Rows)
5     {
6      DataRecordAPI.IDataRecord dr_target =     DataRecordAPI.DataRecordManager.GetDataRecord(dm);
7      foreach (DataColumn c in r.Table.Columns)
8      {
9       string fieldName = c.ColumnName;
10      string fieldValue = r[c.ColumnName.ToString()].ToString();
11      SetDataValue(dr_target, fieldName, fieldValue);
12     }
13     cDR.Add(dr_target);
14    }
15    IEnumerable<DataRecordAPI.IDataRecord> eDR = cDR;
16    DataLinkAPI.IDataSet ds =  DataLinkAPI.DataLinkManager.GetDataSet(eDR);
17    dl.Write(ds);
18   }
```

The Load function builds a list of DataRecordAPI DataRecords and adds them to the DataLinkAPI dataset.

- Line 6 creates a DataRecordAPI DataRecord to be filled and added to the list.
- Line 11 uses an internal function SetDataValue to set the field values on the DataRecord. We will look at that function below.
- Line 16 builds a DataLinkAPI DataSet
- Line 17 uses the BlaiseAgent's DataLinkAPI DataLink and writes that DataLinkAPI DataSet to the Blaise 5 database

This example ignores issues like insuring primary key uniqueness and other errors. But, we will take a quick look at the SetDataValue function to discuss data types. The function receives a value as a string and converts it to the appropriate Blaise data type or non-response attribute for the field.

```
1   private void SetDataValue(DataRecordAPI.IDataRecord dr, string fieldName, string fieldValue)
2     {
3       if (!string.IsNullOrEmpty(fieldName) && !string.IsNullOrEmpty(fieldValue))
4       {
5           DataRecordAPI.IField fld = dr.GetField(fieldName);
6         if (fieldValue == “DK”)
7         {
8            fld.DataValue.SpecialAnswer = MetaAPI.Constants.SpecialAnswerNames.DontKnow;
9         }
10        else
11        {
12           DataRecordAPI.IDataValue dv;
13           if (fld.Structure == DataRecordAPI.FieldStructure.Data)
14           {
15              switch (fld.DataValue.DataType)
16              {
17                 case StatNeth.Blaise.API.DataRecord.DataType.Classification:
18                    if (fld.Definition.Type.IsValid(fieldValue))
19                    {
20                       fld.DataValue.Assign(fieldValue);
21                    }
22                    break;
23                 case StatNeth.Blaise.API.DataRecord.DataType.Date:
24                    dv = fld.DataValue;
25                    dv.DateValue = DateTime.Parse(fieldValue);
26                    fld.DataValue.Assign(dv);
27                    break;
28                 case StatNeth.Blaise.API.DataRecord.DataType.Enumeration:
29                    fld.DataValue.EnumerationValue = int.Parse(fieldValue);
30                    break;
31                 case StatNeth.Blaise.API.DataRecord.DataType.Integer:
32                    fld.DataValue.Assign(fieldValue);
33                    break;

...
```

We can look at a few special cases. (We deleted from the listing more examples of converting some of the Blaise data types that were further examples of setting the field value using the Parse method to convert the string to a corresponding system type.)

- Line 5 creates the DataRecordAPI Field object that will receive the appropriate data type and value
- Lines 6 and 8 test for a received non-response code, e.g. "DK", and set the appropriate Blaise 5 DataRecordAPI Field DataValue.SpecialAnswer.
- Line 12 create a local DataRecordAPI DataValue to hold the type and value
- Line 13 checks that the DataRecordAPI field is a simple field and not block or array, by verifying that the DataRecordAPI Field.Structure is DataRecordAPI.FieldStructure.Data

The following lines perform type conversions by inspecting the DataRecordAPI Field.DataValue.DataType, performing any necessary conversions from the string value, and setting the value with a call to DataRecordAPI Field.DataValue.Assign().

A few might be worth mentioning:

- Line 18 illustrates one of the overloads of the DataRecordAPI Field's Definition.Type.IsValid() method. This example tests a string as a valid classification value. Other overloads of the method validate integers, real, datetime, and arrays of integers for a set type item.
- Line 28 sets the field's DataValue.EnumerationValue to the integer corresponding to that enumerated value

Some things we learned:

- You may need to match the CPU(X86 or x64) to match the version of StatNeth.Blaise.Data.SQLite that Blaise 5 installs

3.2. Display status information – selected cases

The example code populates a SQL data table from the Blaise dataset. Again, for simplicity, it assumes that the table's column names are the same as the Blaise dataset's field names. And, in this case, it displays the status information for a set of cases specified by an incoming SQL data table and the Blaise database's primary key name. (For simplicity, we assume a single key field.)

```
1   public System.Data.DataTable showAllStatus(DataTable dtable, string BlaiseKeyName)
2     {
3       foreach (DataRow dRow in dtable.Rows){
4         string BlaiseKeyValue = dRow.Field<string>(BlaiseKeyName);
5        // Retrieve record based on Primary key
6         DataRecordAPI.IDataRecord dr = GetDataRecord(dm, dl, BlaiseKeyValue);
7         if (dr != null)
8         {
9           foreach (DataColumn cx in dRow.Table.Columns)
10          {
11            string fn = cx.ColumnName;
12            dRow.SetField(fn, ReadDataValue(dr, fn));
13          }
14        }
15      }
16      return dtable;
17    }
```

This example loops through the incoming SQL data table, retrieving the Blaise DataRecordAPI DataRecord by its key value, Line 6 - GetDataRecord(). It then populates a SQL data row with the Blaise field values, converted from the Blaise data type to string, Line 12 – ReadDataValue(). It omits error handling for simplicity.

Looking at GetDataRecord:

```
1  private DataRecordAPI.IDataRecord GetDataRecord(MetaAPI.IDatamodel dm, DataLinkAPI.IDataLink dl,
2  string primaryKey)
3    {
4      DataRecordAPI.IDataRecord dr;
5
6      DataRecordAPI.IKey kv = DataRecordAPI.DataRecordManager.GetKey(dm, "PRIMARY");
7      kv.Fields[0].DataValue.StringValue = primaryKey.ToString();
8      dr = dl.ReadRecord(kv);
9      return dr;
10   }
```

- Line 6 uses the DataRecordAPI DataRecordManager.GetKey method to return Key object from the primary key.
- Line 7 populates the Key object with the requested key value
- Line 8 uses the DataLinkAPI DataLink.ReadRecord method to return the DataRecordAPI DataRecord object with the requested key value.

Once the code has the retrieved record, the BlaiseAgent's ReadDataValue method converts it to a string for the SQL DataTable being built as the return value.

```
1  private string ReadDataValue(DataRecordAPI.IDataRecord dr, string fieldName)
2    {
3      string retValue = "";
4      if (!string.IsNullOrEmpty(fieldName))
5      {
6        DataRecordAPI.IField fld = dr.GetField(fieldName);
7
8        switch (fld.DataValue.AnswerStatus)
9        {
10         case DataRecordAPI.AnswerStatus.SpecialAnswer:
11           retValue = fld.DataValue.SpecialAnswer;
12           break;
13         case DataRecordAPI.AnswerStatus.Response:
14           if (fld.Structure == DataRecordAPI.FieldStructure.Data)
15           {
16             switch (fld.DataValue.DataType)
17             {
18               case StatNeth.Blaise.API.DataRecord.DataType.Classification:
19                 //
20                 break;
21               case StatNeth.Blaise.API.DataRecord.DataType.Date:
22                 retValue = fld.DataValue.DateValue.Value.ToString();
23                 break;
24               case StatNeth.Blaise.API.DataRecord.DataType.Enumeration:
25                 retValue = fld.DataValue.EnumerationValue > 0 ?
                       fld.Definition.Type.Categories.GetItem(fld.DataValue.EnumerationValue).Name : null;
26                 break;
27               case StatNeth.Blaise.API.DataRecord.DataType.Real:
28                 retValue = fld.DataValue.RealValue.ToString();
29                 break;
30               case StatNeth.Blaise.API.DataRecord.DataType.Set:
```

```
31              string value = "";
32              int idx;
33              foreach (var r in fld.DataValue.DynamicValue)
34              {
35                idx = r;
36                value = fld.Definition.Type.MemberType.Categories.GetItem(idx).Name;
37                retValue = string.IsNullOrEmpty(retValue) ? value : retValue + "," + value;
38              }
39              break;
...
```

- Line 6 Like SetDataValue() above, ReadDataValue() operates on a DataRecordAPI Field object. It is retrieved from the incoming DataRecordAPI DataRecord by field name with the GetField method.
- Line 10 initiates a check on the DataRecordAPI Field. DataValue.AnswerStatus to distinguish non-response (DataRecordAPI.AnswerStatus.SpecialAnswer) from a response (DataRecordAPI.AnswerStatus.Response)
- Having determined that the field has its Structure property as DataRecordAPI.FieldStructure.Data, not a block or array (Line 14), the function converts the field depending on its DataValue.DataType.
- In Line 24, an enumerated field (DataType.Enumeration) is converted to its more descriptive Category name from its integer data value using Field.Definition.Type.Categories.GetItem(fld.DataValue.EnumerationValue).Name
- Starting in Line 30, a set field is converted to a comma-separated list of values. It loops through the field's DataValue.DynamicValue, retrieving the integer value, and converts it to its category name.

3.3. Display status information – all cases

To display status for all cases, we can read the entire data set and loop through its records.

To read the entire dataset, or a portion of a dataset with a primary key:

```
1   private  DataLinkAPI.IDataSet  getDataSet(string startKey, int recordLimit, bool includePrimaryKey)
2     {
3       DataLinkAPI.IDataSet ds;
4       string keyName = "";
5       if (dm.Keys.Count >= 1)
6       {
7          DataRecordAPI.IKey key = GetPrimaryKey(dm, startKey);
8          // Read a set of records:
9           ds =  dl.Read(key, DataLinkAPI.ReadOrder.Ascending, recordLimit, includePrimaryKey);
10      }
11      else
12      {
13         // Read all records
14         ds = dl.Read("");
15      }
16      return ds;
17    }
```

- Lines 9 and 14 illustrate two overloads of the DataLinkAPI DataLink object's Read method.

To loop through the resulting DataLinkAPI Dataset, retrieving the DataRecordAPI DataRecord:

```
1   DataLinkAPI.IDataSet ds = getDataSet("", -1, true);
2      if (ds != null && ds.RecordCount > 0)
3      {
4         while (!ds.EndOfSet)
5         {
6            DataRecordAPI.IDataRecord dr = ds.ActiveRecord;
7
8            // ... process the DataRecord
9            ds.MoveNext();
10        }
11     }
```

4. Stress testing

As part of production deployment it is often useful to determine the capacity of the servers to handle a particular instrument and the anticipated user load. Factors affecting capacity include:

- Number of users and expected distribution of users over time
- Size of the data model
- Size and configuration of the server hardware

This section discusses an approach to testing the capacity of a web deployment of a Blaise 5 instrument, using a testing tool that simulates multiple users interacting with the instrument on the anticipated server configuration. Given the variation in instruments, user behavior, and server configuration, it is not a general prediction of how Blaise 5 instruments will perform, but describes some of the considerations and approaches for stress testing a Blaise 5 instrument in a server environment.

4.1. Stress Test Design Considerations

The purpose of stress testing is generally to verify that a given server park configuration will perform adequately at times when the maximum number of concurrent users are active. In a web-based survey, it is very difficult to predict what the maximum number of users will actually be as web surveys are normally self-administered and users take the survey at their convenience. There is also a distinction between the number of users that may be concurrently active in the survey and the processing load of this usage on the server park as a considerable amount of connected time for a user is spent reading questions and typing responses. Although some resources are used to support any connected user, there is a spike in resource utilization once a user submits a page to the server when their data is being processed. Because of this, and in order to not continually over deploy server resources, it is important to configure the test to simulate the expected maximum number of connected users and model the expected profile of their behavior, in terms of wait time between pages, etc.

The stress testing will then help determine what an adequate server park configuration must be to support this expected maximum load and also measure the rate of response degradation as usage is increased beyond the maximum. This information can be used to balance the costs of increased server capacity with the risks of unexpected spikes in concurrent usage.

We used Microsoft's Visual Studio Test Manager 2012, but other tools have similar consideration as those discussed here. The VS2012 Test manager provides the ability to set up a Web Test script by recording a user session against a Blaise 5 web survey. The Web Test script can then be associated with a Load Test, which allows one to specify various parameters like no. of users, test run time, think time, type of load (constant, stepped or simultaneous). Additional scenarios can be added to set different stress levels for the application. A test warm up time can be set to account for any application start up delays.

In our stress testing exercises, we held our server park configuration fixed at one virtual Blaise 5 server plus a SQL Server database server and increased the number of concurrent users from 50 to 200 in 50 user increments to observe the average response time of the survey as the load increased. Except in rare circumstances, perhaps a scripted scheduled training session, it would be unlikely that all 200 users would simultaneously submit a page for processing. And, we expect that users will spend varying amounts of time on each page. We configured the test tool to use a think time at each page randomly distributed around the time we spent on each screen when the script was recorded. To obtain a more realistic arrival time for the pages, and avoid an initial spike of unrealistic simultaneous access, we specified the rate at which users would start the test. Once the test was started and after a short delay we arrived at the specified maximum number of users and maintained that number throughout the test.

We selected a test duration that we thought would allow the users to be on reasonable distribution of pages and locations in the instrument. A longer test duration also allowed us to monitor server resources to see if there was increasing demand on memory (indicating possible leakage) or other resources such as the ASP.Net worker process and to monitor the frequency of application problems, such as page failure, database contention, web exceptions and instrument failures.

In working through various preliminary test results, we varied a number of configuration settings on our one server Server Park such as the number of processors and the amount of memory to see the effect on test results. These configuration changes were somewhat easier to make since the test machine was a virtual server and required no physical hardware change.

We also found that the configuration of the client machines executing the test scripts was an important component of the test design. As these client machines are simulating the execution of a browser-based application for multiple users, it is important that limitations in the configuration of the client test machines do not create artificial bottlenecks that may skew the test results. We found that it was important to add new client test machines for every 75 – 100 simulated users, e.g., simulating 200 users might best be done by using 3 test machines each simulating 70 users. These ratios will be different based on the testing tool and the configuration of the client machines but is an important point to consider.

4.2. Creation of the test script

The testing tool we used provides a feature that records all of the key strokes and timing of a user session through a Blaise 5 instrument. This recorded session can then be "played back" by the test tool to simulate multiple users completing the survey with the same responses. When you record a test script for an instrument, one approach is to provide responses that reflect an "average" path through the instrument, e.g., when it contains a roster that would require looping, select some middle-of-the-road roster sizes. Alternatively, the script can be recorded using responses that reflect the most complex and resource intensive path through the instrument to reflect higher usage profiles.

Typically, each invocation of the instrument requires a case id. Our tool allowed us to build a database of case ids and configure the script to use a unique id for each test cycle. We limited the number of case ids to test a mixture of new and previously opened cases.

4.3. Metrics

Stress testing is normally an iterative process where test scenarios are run and server configurations are changed until an adequate configuration is achieved. It is important when running stress tests to define and record various metrics to establish criteria for a successful test. Measures that we used to evaluate the success of the test included average response time (the time a user waited for a page to be ready for input) and the 95th percentile of response time on any page.

We also monitored several performance counters on the server, some associated with the server resources and some with the ASP.Net framework. These measures can be used to identify resource bottlenecks so that adjustments can be made to the configuration of the Server Park if necessary. Items we tracked included:

ASP.Net Counters;

- Application Restarts
- Requests Queued
- Worker Process Restarts

- Requests per second
- Errors Total

Web Server Processor;

- % CPU Utilization
- Memory Utilization

Database Server;

- % CPU Utilization
- No. of connections

In addition to these performance counters, following key metrics captured by the test tool were used to determine the health of the application under various loads.

- Tests/Sec
- Tests Failed
- Avg. Test Time (sec)
- Pages/Sec
- Avg. Page Time (sec)
- Requests/Sec
- Requests Failed
- Avg. Response Time (sec)

In addition to the automated metric collection, we also had a few live users also navigate the instrument while the load tests were running, noting any unusual delays or problems, and to compare the metrics with the actual user experience, e.g. the average page time recorded by the tool was compared with the average time it took for a page to load during a simultaneous manual test by a user.

4.4. Scaling Blaise 5 server capacity

Blaise 5 offers a number of strategies to scale a Blaise 5 server configuration if it is determined that additional resources are required. These include:

- The resources for any physical/virtual server in a Server Park can be increased to provide additional resources.
- The various server roles can be deployed to different physical/virtual servers.
- All of the server roles except the data server can be installed on 2 or more servers. Blaise 5 will manage the distribution of work across these servers.

4.5. Thoughts on Stress Testing

Stress testing is an important step in preparing for a production implementation of Blaise 5 particularly if a large number of respondents is expected or if a complex survey is used. In this section we discussed a number of considerations and options for conducting stress testing based on our preliminary work in this area.

We plan to continue our work in stress testing Blaise 5 in a server environment. It is our goal to develop some baseline stress testing results that can be used to help organizations begin to project the types of usage loads various server parks configurations can support.

Conclusion

Blaise 5, especially with its new modes of data collection, will require new considerations for configuration, deployment, security, and maintenance. With its unified design across modes of data collection many of these considerations are also unified and may be dealt with a common approach.

Rhoads,Mike and Ray Snowden. “Security Considerations in Blaise Environments: Options andSolutions”. *Proceedings of the 12th International Blaise Users Conference*. September 2010. <http://www.blaiseusers.org/2010/papers/7e.pdf> (August 8, 2013)