Overview

ARCS is an acronym for *Application Resource Control Service*. It is a standard service that is specifically designed for controlling the use of application resources. ARCS fulfills the needs of resource locking, concurrency control, user counting, etc. A resource is anything that two or more entities, such as users, compete for. Examples of a resource include: a user-license, a suite of SQL tables/records, an installation process, a reservation, etc.

ARCS runs on a single Windows machine as an “NT Service”. It meets the standard of “full recovery”, which basically means that it maintains its state through anything short of pulling the power plug (and it would probably even survive that).

Your application may make simple requests to ARCS via HTTP. ARCS is a type of software that is commonly known as a “web service” (though using the SOAP protocol is optional). ARCS can be communicated with over an intranet or the internet.

Neither network file byte-locking nor DBMS table/record locking is utilized. Both of those methodologies have known limitations. ARCS is a standalone service that is ideally suited for resource control and does not utilize or depend on any of the myriad of historical methodologies.

Building such a piece of software yourself may seem reasonable at first glance but it is almost surely the case that it will cost you more money than it takes to purchase ARCS. Matching the quality of ARCS would also take far longer than you may anticipate. Also, a given developer may already know how to use ARCS; if he/she doesn’t, they will be well on their way in 15-20 minutes.

In the future, Littlearth, Inc. will be working on a central/global ARCS that can be utilized by any software that can hit the World Wide Web.

Installation

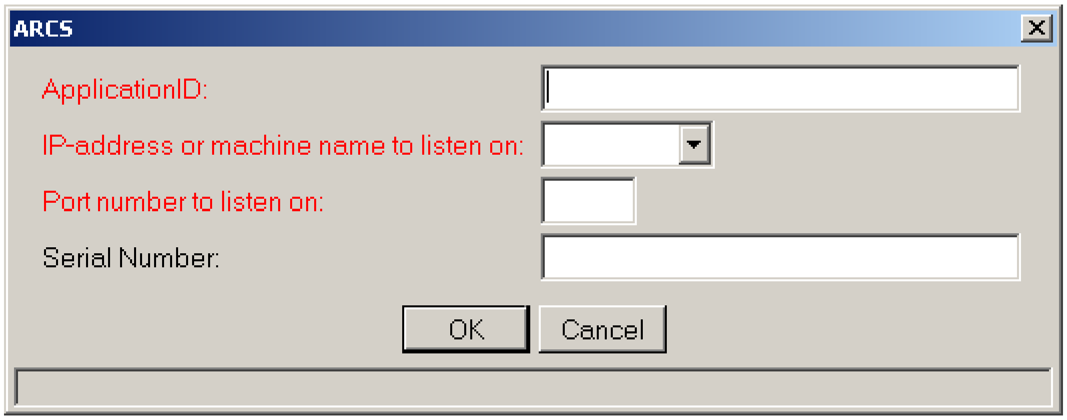
ARCS can be installed as an NT Service on any Microsoft Windows OS version that Microsoft supports. No runtime prerequisites are necessary, e.g. Java Runtime, .NET Framework. No networking software is necessary, e.g. Novell Client, Microsoft Networks Client. No DBMS is necessary, e.g. SQL Server. No webserver-like software is necessary, e.g. IIS.
There are no minimum requirements for processor speed, RAM, etc.

There is no installation required on the client machines (where your application is installed). A client machine can run any type of operating system and it may be an end-user workstation or a server of any kind. The client machine may indeed be a server hosting multiple users itself, e.g. a web server. The client machine simply has to be equipped to make HTTP requests. On Windows machines, this is easily accomplished using a "built-in" COM object such as "MSXML2.ServerXMLHTTP".

You may download the ARCS Setup program at http://www.arcs.us/downloads/Setup.exe. Run Setup.exe on the machine that you wish to have ARCS reside on. This typically takes less than a minute to perform. ARCS may already be installed on the machine, serving other applications. Running Setup ensures that your particular application is registered properly with ARCS.

Upon running Setup.exe, you will be prompted for the following information:

1) ARCS installation directory (unless ARCS is already installed).
2) *ApplicationID*. This identifier insulates sets of users competing for resources. It may be almost anything you like as long as it is distinct from any other *ApplicationIDs* currently installed. This name is also utilized in the URL that is used to access ARCS for your particular application.
3) IP-address and port to listen on.
4) Serial Number. If not specified, then your particular *ApplicationID* can still be used but it is treated as an evaluation (a maximum of 2 users).


ARCS

ApplicationID:

IP-address or machine name to listen on:

Port number to listen on:

Serial Number:

Setup.exe may also be called silently from within another installation program as:

…\Setup.exe <u>silent="Yes"</u>

The following are the other parameters that can be used in the command line:

appdir	- ARCS installation directory (ignored if already installed)
appid	- *ApplicationID*
ipaddress	- IP-address or machine name to listen on
port	- Port number to listen on
serialnumber	- Serial number

Evaluation

All installations of ARCS are free, fully-featured versions. Your particular application may register with ARCS in evaluation-mode or in licensed-mode. Evaluation-mode means that the maximum number of users that can compete for resources (per application) is **two**. This is appropriate for evaluating all of the features and for testing your own code. To allow three or more users per application to compete for resources, an *ApplicationID* license must be purchased (a serial number is issued).

You may download the ARCS Setup program at: http://www.arcs.us/downloads/Setup.exe.

Purchase

To utilize ARCS beyond an evaluation, an *ApplicationID* license must be purchased. An *ApplicationID* is supposed to be associated with a single end-user application, which houses sets of users competing for shared resources. Your *ApplicationID* should be registered on the server machine where ARCS is installed. Such a server likely exists at each site where your end-user application is installed, but it could be a centralized server that is accessible over the internet. There is no limit to how many sites/servers a given *ApplicationID* is registered on. There is no limit to how many developers integrate with ARCS.

A single *ApplicationID* license is priced at 800 USD. Each additional *ApplicationID* license costs 200 USD.

These can be purchased at http://www.arcs.us/purchase.html.

Support

User support is carried out via email with info@arcs.us.
Consulting services are also available.
Unless otherwise specified, all ongoing enhancements and maintenance are included in the original cost of ARCS.

The latest version of ARCS can always be downloaded from http://www.arcs.us/downloads/Setup.exe.

Performance

There are so many variables affecting general performance but just as a very rough ballpark, with today's average machine, ARCS can accommodate several 1000 users, each hitting the service every 5 minutes, on average. That's just for a single application; a single machine can host multiple applications, each potentially getting close to that kind of performance (depending again on many variables and the configuration of the IP-address/port pairs). The architecture allows for a very high level of scalability.
If you can get ARCS to choke under your load, we want to know! info@arcs.us

API

The API (Application Programming Interface) consists of a set of methods each of which are formed in the body of an HTTP request. The applicable URL contains the IP address of the machine that ARCS is installed on as well as your specific *ApplicationID*, e.g. http://192.44.11.181:8080/*MyApp*. (The *ApplicationID* is specified during Setup.) The URL is not case-sensitive. The WSDL document can be found at: http://192.44.11.181:8080/*MyApp*/ARCS.WSDL.

The basic idea is that you begin an ARCS session on behalf of some user of your application. You then bind some specified resource for that user, the user somehow utilizes the resource in your application for some period of time, and then you release the resource when the user is finished with it. While the resource is bound, it cannot be bound by any other user (unless you specify some Capacity greater than 1 for, e.g. user counting). However, when applicable (see Threshold below), the resource can be bound by some other user if the original user hasn't checked-in "recently". Applications do indeed crash and when such is the case, the user should not be allowed to continue binding some resource if another user does actually try to bind that same resource. Keep in mind that a resource may be any abstract thing. To ARCS, it is simply an identifier for something that it aptly "protects".

The following table lists all of the basic method calls and the fundamental arguments. Where X is placed, the argument is required; where {X} is placed, the argument is optional. All method names, argument names, and argument values are case-sensitive.

Arguments / **Method**	UserID	ResourceID	Threshold	Capacity
Begin	X		{X}	
Bind	X	X		{X}
Release	X	X		
CheckIn	X			
Info	{X}	{X}		
End	X			

Arguments

The arguments are described before the methods because the descriptions of the methods make more sense after knowing what all of the arguments mean.

UserID (string)
Identifier of the user.
This is a competitor/owner of resources that are encapsulated by a particular *ApplicationID.*

ResourceID (string)
Identifier of the resource.

Threshold (non-negative integer or float)
This is the amount of time, in seconds, that, if surpassed, the user is susceptible to being wiped out, since it may be assumed that the user's application has crashed. The user is not a candidate for being wiped out until an applicable resource slot is actually needed or a user slot beyond the maximum is needed.
Default: 0 ➔ The user cannot be wiped out.

Capacity (integer)
Maximum number of users allowed to bind/own the resource.
Setting this to a value greater than 1 is useful for, e.g. user license counting.
Setting this to 0 means there is no constraint on the usage of the resource (good for basic feature tracking).
Setting this to a value less than 0 imposes that capacity (its absolute value) on any other user's calls too.
Default: 1 ➔ exclusive access

Methods

Begin - Begin a new session for some user.

Arguments: UserID, {Threshold}
Result: Success

1	Success.
0	The ARCS user-limit had already been reached (need an *ApplicationID* license to get a higher user-limit).

Bind - Bind a resource for some user.

Arguments: UserID, ResourceID, {Capacity}, {Data}
Result: Success

1	Success. If the resource was already bound by the passed user, its update-timestamp was simply updated.
0	The resource could not be bound because it is already being used at full capacity.
-1	The passed UserID does not exist.

Release - Release a resource for some user.

Arguments: UserID, ResourceID
Result: Success

1	Success (was previously bound by the passed user and is now released).
0	The resource was not previously bound by the passed user or by anybody else.
-1	The passed UserID does not exist.

CheckIn - Check-in with the server and update the activity-timestamp for the passed user. All other methods update the activity-timestamp as well. This method is typically called on a timer (in your application) having an interval that is less than or equal to the threshold passed to **Begin**. Basically, it is called to show that the user's application hasn't crashed; this is intended to ensure that the user session is not wiped out upon some other user attempting to bind a resource this user has bound. A server hosting multiple users, e.g. a web server, may call **CheckIn** on behalf of its users when such users check-in with it.

Arguments: UserID
Result: Success

1	Success. The activity-timestamp was updated.
-1	The passed UserID does not exist.

Info – Retrieve information about some user and the resources for that user.

Arguments: {UserID}, {ResourceID}

If no UserID is passed or it passed as an empty string, it is implied to be all of the current users.

If no ResourceID is passed or it is passed as an empty string, it is implied to be all of the current resources.

Result: There are three elements in the result.

[1]	Current timestamp of the system on the server. This may be compared to your own timestamp or to the timestamps returned in the 2nd and 3rd elements below (the client may be in a different time zone than the server). [See the section "Timestamps:" for the format.]
[2]	This element contains information for each user that satisfies all of the following criteria: A) Matches the UserID passed B) Is binding the ResourceID passed Its exact structure varies by protocol but it is abstractly defined as rows and columns of a table. There is one row per user and the columns are as follows: UserID, InitialTimestamp, ActivityTimestamp, Threshold.
[3]	This element contains information for each resource that satisfies all of the following criteria: A) Matches the ResourceID passed B) Is bound by the UserID passed Its exact structure varies by protocol but it is abstractly defined as rows and columns of a table. There is one row per resource and the columns are as follows: UserID, ResourceID, InitialTimestamp, UpdateTimestamp, Capacity, Data.

End – End the session for some user.

Arguments: UserID

Result: Success

1	Success. All of the user's bindings on resources were released.
-1	The passed UserID does not exist.

The following methods are adjuncts to the base API. They are very useful for storing and retrieving data, e.g. session state data across a server farm. They also follow the paradigm of having to run **Begin** and optionally **End**. They work as a suite and are useful for storing arbitrary data of any kind. ARCS places no constraint at all on what can be stored. For a given UserID/DataID, whatever it receives via **Store**, is exactly what is returned via **Retrieve**. Note that many HTTP tools have difficulty sending arbitrary binary data, even when the HTTP content-type is appropriately set to "application/octet-stream". Typically, it is the null character that poses problems for these tools. Converting to base64 is a good solution to this.

Store – Store any data for some user and an arbitrary identifier for the data.

Arguments: UserID, DataID, Data

Result: Success

1	Success.
0	The data could not be stored for some reason (should "never" happen).
-1	The passed UserID does not exist.

Retrieve – Retrieve data that was previously stored via **Store**.

Arguments: UserID, DataID

Result: There are 2 elements in the result.

[1]	1 – Success 0 - DataID does not exist -1 - The Passed UserID does not exist.
[2]	Data (empty if first element is 0 or –1)

Remove – Remove some data that was previously stored via **Store**. Upon calling **End**, all data associated with the given user, is removed.

Arguments: UserID, DataID

Result: Success

1	Success (was previously stored by the passed user and is now removed).
0	The data was not previously stored by the passed user.
-1	The passed UserID does not exist.

Example - Sequence of Steps

The following table exemplifies a typical simple sequence of steps.
Where a cell contains Request or Response, it is to be understood that the respective content is for the body of the HTTP sent or received.
In this example, the resource referred to as *MyUserLicense* is meant to be your application's idea of a user-license; it is not referring to user licensing with regard to ARCS itself.

1) Request: **Begin** MyUserA 600 Response: 1	**2)** Request: **Bind** MyUserA *MyUserLicense* 20 Response: 1	**3)** Miscellaneous application activity …
4) *[Probably called several times throughout the user session, on a timer]* Request: **CheckIn** MyUserA Response: 1	**5)** Request: **Bind** MyUserA *MyResourceN* Response: 1	**6)** Edit *MyResourceN* …
7) Request: **Release** MyUserA *MyResourceN* Response: 1	**8)** Miscellaneous application activity …	**9)** *[This step is optional since all resources will be released when **End** is called in the next step.]* Request: **Release** MyUserA *MyUserLicense* Response: 1
10) Request: **End** MyUserA Response: 1		

Protocols

There are 3 different protocols that can be used in the requests/responses in the HTTP body. They all have the same breadth of functionality and behavior. Use the one that is most convenient for you. Contact info@arcs.us if you have a need for a reasonable protocol that is not listed below.

1) ***Basic*** (note: The example above was written using this protocol.)
Use the following HTTP content-type: *text/plain*

[Request]:
Each line should be separated by the following two characters: carriage-return, line-feed (or either one of them singly). The first line contains the method name. All subsequent lines contain the arguments to the method, in the order specified by the Methods/Arguments table at the top of this section. Optional arguments can simply be omitted since they are, by design, at the end. For the method **Store**, only carriage-return may be used as the delimiter (the data itself may contain anything at all). Similarly, the result of **Retrieve** is delimited only by carriage-return.

[Response]:
For all methods except for **Info** and **Retrieve**, the numeric result is on a single line. **Retrieve** has its 2 elements delimited by the carriage-return character. For **Info**, see the detailed explanation at the end of this section.

2) ***SOAP***
Use the following HTTP content-type: *text/xml*

[Request]:
The request is a standard SOAP wrapper and body. Below is an example using the method **Begin**. Other methods have an analogous format in that the names of the elements follow the argument names in the Methods/Arguments table at the top of this section.

```
<?xml version="1.0"?><SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><SOAP-ENV:Body>
        <Begin>
                <UserID>MyUserA</UserID>
                <Threshold>600</Threshold>
        </Begin>
</SOAP-ENV:Body></SOAP-ENV:Envelope>
```

[Response]:

```
<?xml version="1.0"?><SOAP-ENV:Envelope …<SOAP-ENV:Body>
        <BeginResponse><Success>1</Success></BeginResponse>
</SOAP-ENV:Body></SOAP-ENV:Envelope>
```

3) ***FormPost***
Use the following HTTP content-type: *application/x-www-form-urlencoded*

[Request]:
The request is encoded as name/value pairs (as specified for the HTTP content-type above). Below is an example using the method **Begin**. Other methods have an analogous format in that the names of the elements follow the argument names in the Methods/Arguments table at the top of this section.

Method=**Begin**&<u>UserID</u>=**MyUserA**&<u>Threshold</u>=600

[Response]:
1

Timestamps

For all of the protocols, timestamps are of the following format, consisting of 23 characters:

```
            ccyy-mm-ddThh:mm:ss.sss
Example:    2006-02-19T20:34:41.434
```

This is a valid format of the XML Schema "dateTime" primitive datatype.

More on the **Info** method

It may be helpful to refer to the description of the result of **Info** under the section Methods.

Applicable to the ***Basic*** and ***FormPost*** protocols: The result elements (the three elements that are described above under Methods) are delimited from each other by two sets of carriage-return/line-feed (4 characters in total).
The second and third elements each delimit their respective rows with carriage-return/line-feed and delimit their respective columns with a tab character. It is recommended that you try it out for yourself to see how the result looks.

The ***SOAP*** protocol can be well understood by following the example below (the SOAP wrapper is omitted, for clarity).

Example - **Info** method using the ***SOAP*** protocol:

[Request]:

```
<Info>  <!-- Retrieve information on MyUserA and all of its resources -->
        <UserID>MyUserA</UserID>
        <ResourceID></ResourceID>
</Info>
```

[Response]:

```
<InfoResponse>
        <SystemTimestamp>
                2006-02-19T34:24:21.534
        </SystemTimestamp>
        <Users>
                <User>
                        <UserID>MyUserA</UserID>
                        <InitialTimestamp>2006-02-19T20:34:41.434</InitialTimestamp>
                        <ActivityTimestamp>2006-02-19T30:34:51.531</ActivityTimestamp>
                        <Threshold>600</Threshold>
                </User>
        </Users>
        <Resources>
                <Resource>
                        <UserID>MyUserA</UserID>
                        <ResourceID>MyUserLicense</ResourceID>
                        <InitialTimestamp>2006-02-19T20:35:02.357</InitialTimestamp>
                        <UpdateTimestamp>2006-02-19T20:35:02.357</UpdateTimestamp>
                        <Capacity>10</Capacity>
                        <Data>whatever you want</Data>
                </Resource>
                <Resource>
                        <UserID>MyUserA</UserID>
                        <ResourceID>MyResourceN</ResourceID>
                        <InitialTimestamp>2006-02-19T23:53:21.333</InitialTimestamp>
                        <UpdateTimestamp>2006-02-19T23:53:21.333</UpdateTimestamp>
                        <Capacity>1</Capacity>
                        <Data>…</Data>
                </Resource>
        </Resources>
</InfoResponse>
```

Example – Simplified Code

You should be able to copy/paste the following pseudo-code and make the necessary adjustments to get something working very quickly.

```
url = “http://192.44.11.181:8080/MyApp”
http = new ActiveXObject("Microsoft.XMLHTTP")

user = “UserA”
threshold =  600
request = “Begin” + “\n” + user + “\n” + threshold

http.open("POST", url, false)
http.setRequestHeader("Content-Type", “text/plain”)
http.send(request)
response = http.responseText // you should then check on the success of this result

request = “Bind” + “\n” + user + “\n” + “MyResourceN”
http.open("POST", url, false)
http.setRequestHeader("Content-Type", “text/plain”)
http.send(request)
response = http.responseText // you should then check on the success of this result
```

Usage Notes

It may be helpful to consider a few helpful notes on how your application may interact with ARCS. Your application may warrant some of the following scenarios or some variation of them.

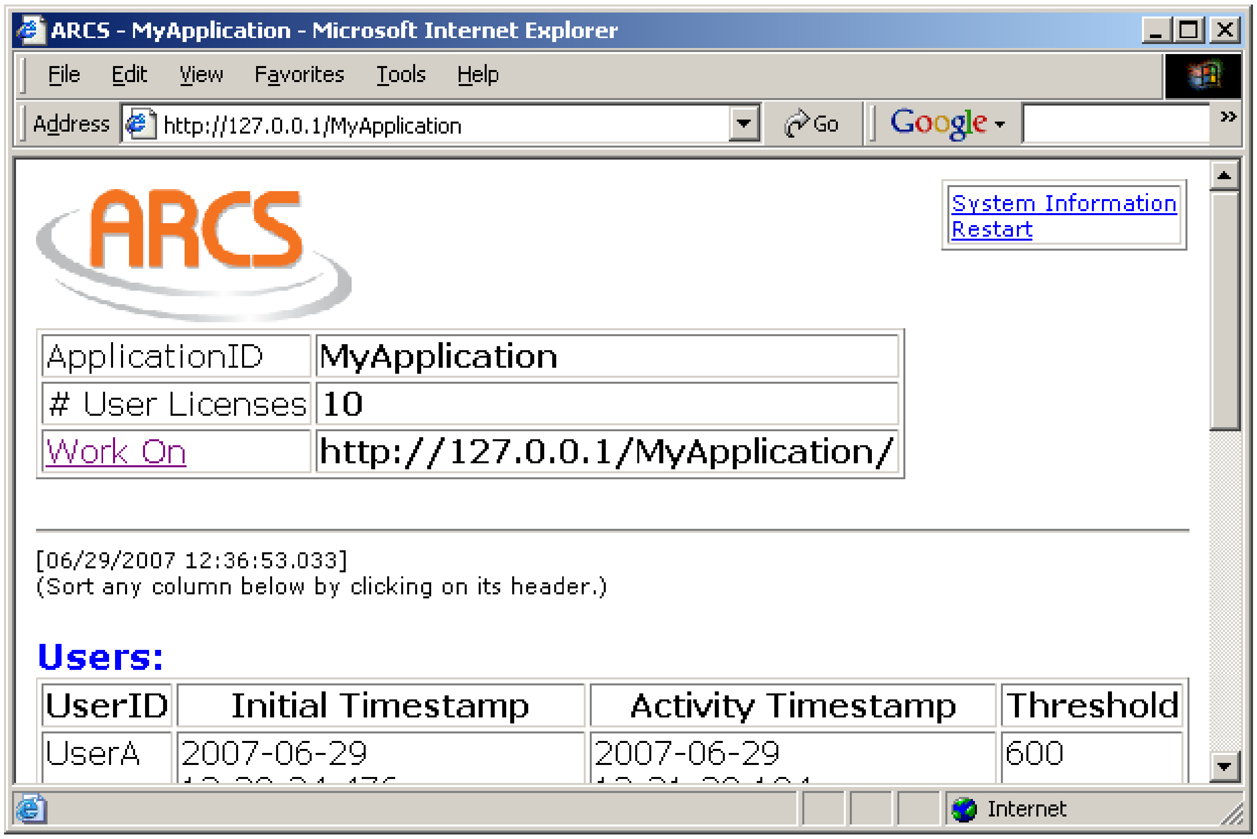
- If a given user for a given application can launch multiple application sessions, you may want to distinguish between those "users" by adding a unique identifier to each of the users, in order to form appropriately unique UserIDs. This will ensure that those user sessions are also in competition with one another.
- If your application crashes and it can somehow detect that the next time it is launched for the same user that crashed, you could call **CheckIn** and if it returns 1, you probably wouldn't want to call **Begin**, since it is apparent that all of your previous resources are still appropriately bound. (**Begin** always starts completely anew and so may or may not be appropriate.)
- Even if you already have a given resource bound, when you're about to write data to your application's database, it is prudent to call **Bind** first in order to see if you still have the resource bound (suppose your application hadn't run **CheckIn** because it lost connectivity for too long and then some other user came along and was able to steal the resource).

  For a write that takes so long that the threshold could theoretically be surpassed during the write, it would also be a good idea to call **Bind** just before the actual writing.

  Even when it is the case that a given resource is not bound, a call to **Bind** before writing is prudent. Upon receiving a successful result from **Bind**, you should check that the data you originally read matches what is currently in the database before committing the write.
- Notice that it is actually possible to steal a resource from some other user (by first calling **Release** for the applicable UserID and ResourceID). While this may not be common practice, it may be warranted by a given type of application.

Monitoring

There is a web browser interface that is useful for monitoring all users and resources. There is also a simple way of testing the API from a web browser (2nd screenshot below). The JavaScript code used in that web page is worthy of review.



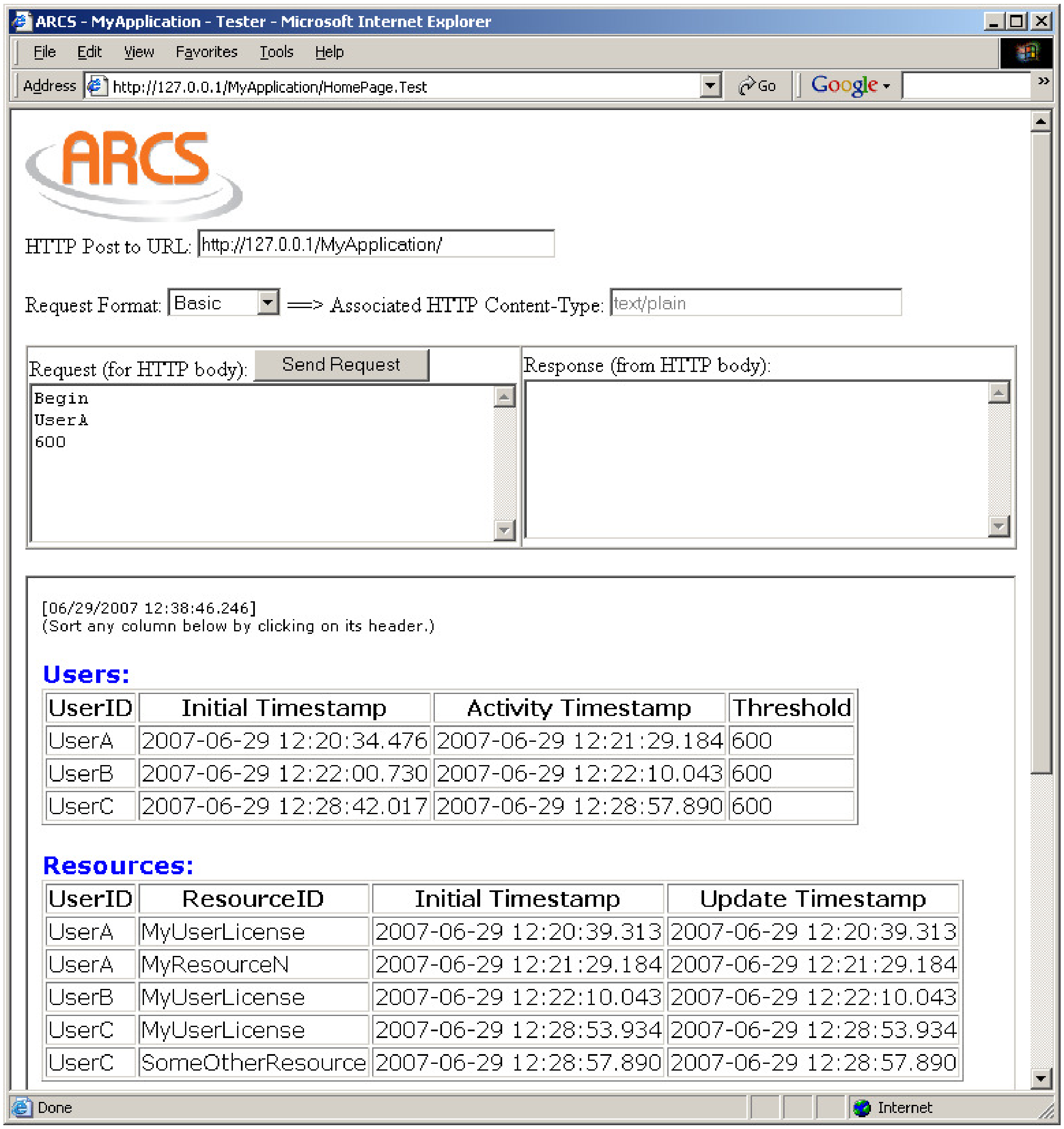
ARCS - MyApplication - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://127.0.0.1/MyApplication
Go
Google
System Information
Restart
Internet

ApplicationID	MyApplication
# User Licenses	10
Work On	http://127.0.0.1/MyApplication/

[06/29/2007 12:36:53.033]
(Sort any column below by clicking on its header.)

Users:

UserID	Initial Timestamp	Activity Timestamp	Threshold
UserA	2007-06-29	2007-06-29	600


ARCS - MyApplication - Tester - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://127.0.0.1/MyApplication/HomePage.Test
Go
Google
ARCS
Send Request
Done
Internet

HTTP Post to URL: http://127.0.0.1/MyApplication/

Request Format: Basic ==> Associated HTTP Content-Type: text/plain

Request (for HTTP body):

```
Begin
UserA
600
```

Response (from HTTP body):

[06/29/2007 12:38:46.246]
(Sort any column below by clicking on its header.)

Users:

UserID	Initial Timestamp	Activity Timestamp	Threshold
UserA	2007-06-29 12:20:34.476	2007-06-29 12:21:29.184	600
UserB	2007-06-29 12:22:00.730	2007-06-29 12:22:10.043	600
UserC	2007-06-29 12:28:42.017	2007-06-29 12:28:57.890	600

Resources:

UserID	ResourceID	Initial Timestamp	Update Timestamp
UserA	MyUserLicense	2007-06-29 12:20:39.313	2007-06-29 12:20:39.313
UserA	MyResourceN	2007-06-29 12:21:29.184	2007-06-29 12:21:29.184
UserB	MyUserLicense	2007-06-29 12:22:10.043	2007-06-29 12:22:10.043
UserC	MyUserLicense	2007-06-29 12:28:53.934	2007-06-29 12:28:53.934
UserC	SomeOtherResource	2007-06-29 12:28:57.890	2007-06-29 12:28:57.890