SDL BMS:
A Simple Broadcast Message Server

Daniel J. Barrett
Software Development Laboratory
Computer Science Department
University of Massachusetts

Arcadia Document UM–93–03
December 20, 1993

Abstract

BMS is a simple, general, broadcast-based message server which is used for communication between C and Ada programs. This document describes its features and usage, including instructions on how to build client programs that communicate via **BMS**.

Contents

1 Introduction

BMS (Broadcast Message Server) is a general message-routing service which provides simple broadcast of message strings among a set of client programs written in C or Ada. It can also invoke other programs on request. **BMS** has been used to integrate the SDL PIC [4] demo and the TESS [2] user interface.

BMS runs under SunOS 4.x and is implemented on top of **Q** [3] from the University of Colorado.

BMS is unrelated to the program of the same name which is a part of HP SoftBench [1].

2 BMS Features

BMS supports two services:

1. Simple broadcast of a message among a set of programs called *BMS clients*.
2. Invocation of other programs.

Programs that wish to use **BMS**'s features must *register* to use them, thereby becoming **BMS** *clients*. Any client may use **BMS** to broadcast a message to all other **BMS** clients or invoke a program. Also, every client is provided with a message queue to receive all messages broadcast while they are registered. When a client is no longer interested in using **BMS**'s services, it may *unregister*.

BMS is very simple and does no filtering of messages. That is, every message is broadcast to every registered client, even to the client that sent the message. **BMS** has no finer-grained delivery method than this. (See Figure 1.) Clients are responsible for inspecting all messages as they arrive to determine their relevance.

Messages are ordinary strings. **BMS** imposes no structure on them. It is the clients' responsibility to agree on a message format that they can all understand.

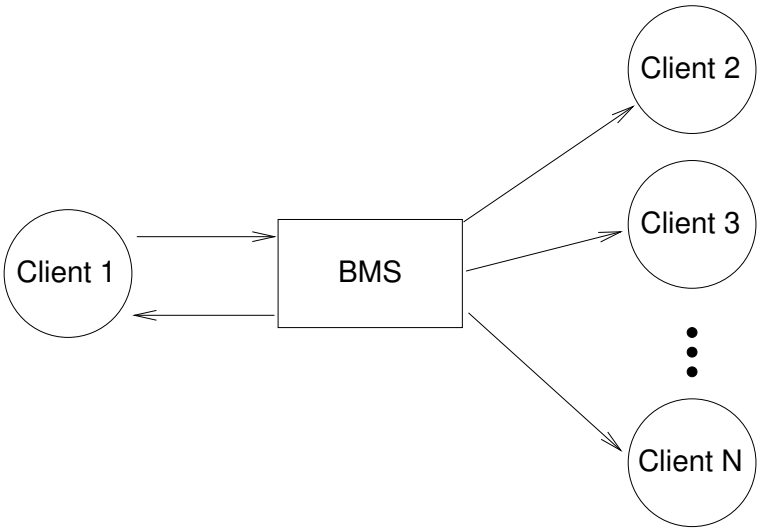

Figure 1: A typical BMS broadcast from Client 1 to all clients.

3 BMS Behavior

The following message delivery properties are true at all times while **BMS** is running:

1. Every client is guaranteed to receive all messages, and only those messages, that are broadcast between the times that the client registers and unregisters.

   (Of course, if a client chooses not to examine its entire incoming message queue, then it will not see all of its messages. If one client needs to know that another has actually processed a particular message, handshaking between the clients will be required.)

2. Messages are delivered strictly in the order that they are received by **BMS**. That is, for all clients c_1 and c_2, if **BMS** receives a message from c_1 before it receives one from c_2, then every client will receive c_1's message before it receives c_2's message.

BMS's semantics of program invocation are:

1. **BMS** invokes a program when sent a string containing a UNIX shell command.

2. **BMS** keeps track of which programs it has invoked that are still executing, by storing the command strings used to invoke the programs.

3. **BMS** will allow only one invocation of a given command string to be running at any time. If a command string is given to **BMS** for invocation, and it exactly matches a that of currently executing program, **BMS** will ignore the new command string.[1]

4. When invoking a command, **BMS** treats all whitespace characters in the command as word delimiters. This is true even if the whitespace is contained inside quotes. So an invoked command such as

   ```
   emacs "my filename with spaces"
   ```

   is treated as having five arguments:

   ```
   emacs
   "my
   filename
   with
   spaces"
   ```

   The program invocation feature of **BMS** was written under time pressure, so a quoted argument parser was not implemented. This was considered unimportant since none of the files in the PIC demo had whitespace in their names.

4 Creating Clients

In order to make use of **BMS**'s services, an Ada or C program will:

1. Register. **BMS** gives the client a registration ID which must be used in all future contact with **BMS**, until after the client has unregistered. (Required.)

2. Create outgoing messages and fill them with data, broadcast them via **BMS**, and destroy them. (All optional.)

[1] This behavior is a requirement for the PIC Demo: at most one invocation of each command may be running at a given time. It was intended that a duplicate command would cause the already-running program's interface to be brought to the front of the display. However, this is not implemented.

3. Ask **BMS** to invoke programs. (Optional.)

4. Receive messages. (Optional.)

5. Unregister. (Required.)

To build such a program, one must:

1. Compile with the needed packages (for Ada) or header files (for C).

2. Link with the **BMS** library.

Instructions for creating Ada and C clients are below.

4.1 Ada Clients

4.1.1 Compiling and Linking

Your client must "with" the **BMS** client package:

```
with BMS_Client;
```

To allow the Ada compiler to locate this package and other important files, add the libraries

```
/u/zoo/arcadia/Q/build.sun4-sunada1.1
/u/zoo/sdl/projects/bms/ada
```

to your library search path.

When linking your client, you will need the **Q** library. Use the linker (`a.ld`) flags:

```
-L/u/zoo/arcadia/Q/build.sun4-cc
-lQ
```

4.1.2 Registration and Unregistration

First, declare variables to contain your client's unique **BMS** registration ID, and one or more messages:

```
id      : BMS_Client.BMS_ID;
message : BMS_Client.BMS_MSG;
```

To register:

```
id := BMS_Client.StartSession;
if id < 0 then
        Put_Line("Could not contact BMS.");
else
        Put("I am registered as client ");
        Put(id, width => 0);
        Put_Line(".");
end if;
```

Registration may fail for several reasons, all of which cause a negative value to be returned by `StartSession()`:[2]

[2]The author recognizes that the use of return codes instead of exceptions is not good Ada style. However, return codes were used for uniformity in the C and Ada interfaces, since C does not support exceptions.

id value	Meaning of error
BMS_START_NO_IDS	**BMS** has no more registration ID's left. The limit is statically defined as the constant MAX_BMS_REGISTERED in the file bms-interface.h. Currently, the value is (arbitrarily) 100.
BMS_START_NO_ENV	You forgot to define a value for the environment variable BMS_CODE_NUMBER. See Section 5, page 15.
BMS_START_NO_POSITIVE	You made an error defining the environment variable BMS_CODE_NUMBER: it must be a positive number. See Section 5, page 15.
BMS_START_NO_SERVER	**BMS** could not be contacted. Is it running? See Section 5, page 15.
BMS_START_ILLEGAL_PROTOCOL	You specified an unknown communications protocol in your BMS_PROTOCOL environment variable. See Section 5, page 15.

To unregister:

```
BMS_Client.EndSession(id);
```

4.1.3 Creating and Destroying Messages

To create a message,

```
str : string(1 .. BMS_Client.BMS_MAX_STRING_SIZE);
len : natural;
message : BMS_Client.BMS_MSG.

Put("Give me a message to send: ");
Get_Line(str, len);
message := BMS_Client.CreateMessage(
                  BMS_Client.BMS_COMMAND_BROADCAST,
                  str(1..len));
if message = BMS_Client.BMS_MSG_NULL then
```

```
        Put_Line("Could not create a message.");
else
        Put_Line("I created a message.");
end if;
```

To destroy a message after you no longer have a need to send it,

```
BMS_Client.DestroyMessage(message);
```

The rules of message creation and destruction are:

1. `CreateMessage()` and `DestroyMessage()` must appear in matched pairs. The two corollaries are:
   - ALWAYS destroy (eventually) all messages that you create. If you assign the value of `CreateMessage()` to the same variable twice in a row, you'll lose the first message and waste memory.
   - NEVER destroy a message that you yourself did not create with `CreateMessage()` (for example, that you obtained by calling `GetNextMessage()`). You will crash.
2. Once a message is created by `CreateMessage()`, it may be safely used (that is, passed to `SendMessage()`) any number of times before you destroy it with `DestroyMessage()`.

4.1.4 Sending Messages

To send a message (after creating it),

```
if not BMS_Client.SendMessage(id, message) then
        Put_Line("I FAILED to send the message.");
else
        Put_Line("I sent the message successfully.");
end if;
```

4.1.5 Receiving Messages

To check whether a message is waiting for you, without removing it from the queue:

```
if BMS_Client.MessageWaiting(id) then
        Put_Line("There is a message waiting.");
else
        Put_Line("There are NO messages waiting.");
end if;
```

To receive the next message in your message queue:

```
message := BMS_Client.GetNextMessage(id);
if message = BMS_Client.BMS_MSG_NULL then
        Put_Line("I could not find a message!");
else
        Put_Line("I got a message: " &
               BMS_Client.BMS_DATA(message));
end if;
```

It is not necessary to call `MessageWaiting()` before `GetNextMessage()`, since `GetNextMessage()` will return `BMS_MSG_NULL` if there are no messages waiting. For example, to receive all of your queued messages:

```
loop
        message := BMS_Client.GetNextMessage(id);
        DoSomethingWith(message);
        exit when message = BMS_Client.BMS_MSG_NULL;
end loop;
```

Messages returned by `GetNextMessage()` are not owned by the client, and must not be destroyed by `DestroyMessage()`.

4.1.6 Invoking Programs

To invoke a program, first create an invocation message,

```
Put("Give me a program to invoke (e.g., xterm): ");
Get_Line(str, len);
message := BMS_Client.CreateMessage(
                        BMS_Client.BMS_COMMAND_INVOKE,
                        str(1 .. len));
if message = BMS_Client.BMS_MSG_NULL then
        Put_Line("Could not create the message.");
else
        Put_Line("I created an invoking message.");
end if;
```

and then send it to **BMS**,

```
if not BMS_Client.SendMessage(id, message) then
        Put_Line("FAILED to invoke the program.");
else
        Put_Line("Tried to invoke the program.");
end if;
```

This message must eventually be destroyed by `DestroyMessage()`.

4.1.7 A Complete Example Client

A complete example is found in the file:

```
/u/zoo/sdl/projects/bms/ada/sample-client.a
```

4.2 C Clients

4.2.1 Compiling and Linking

Your client must include the **BMS** header file:

```
#include "bms.h"
```

To allow your C compiler to locate this file, use the flag:

```
-I/u/zoo/sdl/projects/bms/c
```

typically as part of your `CFLAGS` macro in your Makefile. When linking your program, you must link with the **BMS** and **Q** libraries. Use the linker flags

```
-L/u/zoo/sdl/projects/bms/c
-L/u/zoo/arcadia/Q/build.sun4-cc
-lbms
-lQ
```

(Note: some C compilers have trouble with multiple `-L` flags. If you use the above flags and some libraries are not found, even though you are sure that the libraries exist, you may have encountered this bug. To work around it, specify the complete pathname of one of the libraries, rather than using the `-l` syntax.)

4.2.2 Registration and Unregistration

First, declare variables to contain your client's unique **BMS** registration ID, and one or more messages:

```
BMS_ID id;
BMS_MSG message;
```

To register:

```
id = BMS_StartSession();
if (id < 0)
        printf("Could not contact BMS.\n");
else
        printf("I am registered as client %d\n", id);
```

Registration may fail for several reasons, all of which cause a negative value to be returned by `BMS_StartSession()`. These values are given in the table in Section 4.1.2, page 7.

To unregister:

```
BMS_EndSession(id);
```

4.2.3 Creating and Destroying Messages

To create a message,

```
message = BMS_CreateMessage(BMS_COMMAND_BROADCAST,
                    "This is my message");
if (!message)
        printf("Could not create a message.\n");
else
        printf("I created a message.\n");
```

To destroy a message after you no longer have a need to send it,

```
BMS_DestroyMessage(message);
```

Important rules (semantics) for creating and destroying messages are given in Section 4.1.3, page 9. The semantics are very similar to those of the standard C functions `malloc()` and `free()`. This section is mandatory reading!

4.2.4 Sending Messages

To send a message (after creating it),

```
if (!BMS_SendMessage(id, message))
        printf("I FAILED to send the message.\n");
else
        printf("I sent the message successfully.\n");
```

4.2.5 Receiving Messages

To check whether a message is waiting for you, without removing it from the queue:

```
if (BMS_MessageWaiting(id))
        printf("There is a message waiting.\n");
else
        printf("There are NO messages waiting.\n");
```

To receive the next message in your message queue:

```
message = BMS_GetNextMessage(id);
if (!message)
        printf("I could not find a message!\n");
else
        printf("I got a message: <%s>\n",
               BMS_DATA(message));
```

It is not necessary to call the function `BMS_MessageWaiting()` before `BMS_GetNextMessage()`, since `BMS_GetNextMessage()` will return `NULL` if there are no messages waiting. For example, to receive all of your queued messages:

```
while ((message = BMS_GetNextMessage(id)) != NULL)
        DoSomethingWith(message);
```

Messages returned by `BMS_GetNextMessage()` are not owned by the client, and must not be destroyed by `BMS_DestroyMessage()`.

4.2.6 Invoking Programs

To invoke a program, first create an invocation message,

```
message = BMS_CreateMessage(BMS_COMMAND_INVOKE,
                    "xterm");
if (!message)
        printf("Could not create the message.\n");
else
        printf("I created an invoking message.\n");
```

and then send it to **BMS**,

```
if (!BMS_SendMessage(id, message))
        printf("BMS FAILED to invoke an xterm.\n");
else
        printf("BMS tried to invoke an xterm.\n");
```

This message must eventually be destroyed by `BMS_DestroyMessage()`.

4.2.7 A Complete Example Client

Complete examples are found in the files:

```
/u/zoo/sdl/projects/bms/c/sample-client.c
/u/zoo/sdl/projects/bms/c/little-client.c
```

5 Running BMS

Each user (really, each user ID) may run at most one invocation of **BMS** at any given time. If several **BMS**'s are run under the same user ID, then any **BMS** clients run under that user ID will register nondeterministically with one **BMS** or the other. This behavior is probably not desirable.

However, multiple **BMS**'s may run on the same machine, provided they are being run by different users. Client programs run by a given user will communicate only with the **BMS** run by that user. These limitations are inherent in the implementations of **BMS** and **Q**.

There are three steps to running **BMS** that must be done before any clients may register:

1. Each user must choose a unique *code number* that identifies his/her invocation of **BMS**. It is accomplished by defining a positive integer value for the environment variable `BMS_CODE_NUMBER`. For example, if you use the Korn shell:

   ```
   $ BMS_CODE_NUMBER=13579
   ```

   Once you have defined this number, you can forget it — its use is completely invisible to the clients.

2. Optionally choose a *communications protocol* for **BMS** to use: either TCP or UDP. (If you don't know what these are, then ignore this item.) This is done by defining the environment variable `BMS_PROTOCOL` to be either `TCP` or `UDP`. For example, if you use the Korn shell:

   ```
   $ BMS_PROTOCOL=TCP
   ```

   By default (if no environment variable is defined), UDP is used.

3. Invoke **BMS** by typing:

```
$ bms &
```

BMSis now running and ready to accept registrations and provide services to clients.

6 Limitations

Due to various design decisions, **BMS** usage is restricted in the following ways:

1. Because **BMS** is built on top of **Q**, it runs only under SunOS and supports only C and Ada clients.
2. **BMS** and all its associated clients must be invoked under the same UNIX user id. See Section 5, page 15.
3. When invoking a program, **BMS** treats all whitespace characters in the command as word delimiters, even if the whitespace is contained within quotes. See Section 3, page 4, item 4.
4. When asking **BMS** to invoke a program, there is no way for the client to know whether the invocation was successful or not. (However, if the program itself happens to register as a **BMS** client, some kind of handshake could be implemented.)
5. **BMS** provides no system-friendly method for clients to wait for incoming messages. Clients may busy-wait or use facilities provided by other programming libraries (such as window systems).

7 Extensions

Barbara Lerner has written an Ada package called `BMS_Interface` that provides a handshake routine and other features (raising exceptions instead of using error codes, support for point-to-point communication, filtering of messages sent by oneself).

References

[1] Colin Gerety. HP SoftBench: A New Generation of Software Development Tools. SoftBench Technical Note Series SESD-89-25, Revision 1.4, Hewlett-Packard, November 1989.

[2] Barbara Staudt Lerner. Principles of Type Evolution. Design document for TESS, in progress.

[3] M. Maybee, L. J. Osterweil, and S. D. Sykes. Q: A Multi-lingual Interprocess Communications System for Software Environment Implementation. Technical Report CU-CS-476-90, University of Colorado, Boulder, June 1990. Currently undergoing review for publication in Software Practice and Experience.

[4] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. Ada-Based Support for Programming-in-the-Large. *IEEE Software*, 2(2):58–71, March 1985.