

Remote debugging extension of the Mono Debugger (MDB)

Date/Version: 2008-03-12

TU Vienna, ICT

Harald Krapfenbauer

<krapfenbauer at ict dot tuwien dot ac dot at>

Dominik Ertl

<ertl at ict dot tuwien dot ac dot at>

Content

1. Overview	3
2. Basic concept of the Mono debugger	3
3. Remote procedure calls.....	4
4. Directory structure	4
5. Client and server code.....	4
6. Description of mdbserver.....	5
7. Input to the inferior application	5
8. Command-line arguments of the Mono Debugger.....	6
9. Concurrent debugging – Ports assignment	7
10. Resources needed on host and target.....	8
11. Installation.....	8

1. Overview

The remote debugging extension for the Mono debugger targets small/embedded systems where the main memory is limited and running the Mono debugger is not feasible.

This extension provides debugging a Mono application running on a remote system from a local terminal. The Mono process executing the debugger runs on the local machine whereas on the remote machine no Mono process is required for the debugger. Input and output to/from the remote application are automatically handled.

From the user's point of view, the operation of the Mono debugging is not affected, except of additional command line arguments. Local debugging is still possible.

Currently, only i386 machines are supported.

2. Basic concept of the Mono debugger

Since the debugged Mono application (called „inferior“) runs in an own process, the Mono debugger uses the following common Linux kernel functionality to deal with the inferior:

- The **ptrace()** system call:
It provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing.
- Reading from files of the proc file system:
The files `/proc/[PID]/mem`, `/proc/[PID]/exe`, `/proc/[PID]/cwd`, `/proc/[PID]/cmdline`, `/proc/[PID]/task/*` from the inferior process are read.
- **waitpid()** function
- **kill()** function
- **fork()** function
- **syscall()** function
- The bfd library uses standard file manipulation functions to read out symbol information: **open()**, **fseek()**, **fread()**...

3. Remote procedure calls

To extend the Mono debugger with remote debugging capability, the Linux system calls listed above are „forwarded“ to the target system, including all necessary parameters, and the results are transferred back to the host system.

Remote execution is realized by means of the Remote Procedure Call (RPC) protocol (<http://en.wikipedia.org/wiki/Rpc> gives a good introduction).

Remote procedure calls were added to the Mono debugger's backend library called **libmonodebuggerserver** which consists of the Mono debugger backend code written in C and the libraries **bfd**, **libiberty** and **opcodes**. The Mono debugger is the RPC client.

For the embedded target where the inferior runs, a tiny server application called **mdbserver** was implemented in C which is the RPC server. It provides all RPC functions that are necessary for remote debugging via TCP sockets.

No Mono instance is required to run **mdbserver** since it is compiled to native code.

Note that usually RPC requires the portmap service (<http://linux.die.net/man/8/portmap>) on the RPC server machine. However, the remote debugging extension of the Mono Debugger uses direct TCP communication, hence the portmap service is not required.

4. Directory structure

<u>debugger</u> :	The debugger root directory
<u>debugger/mdbserver</u> :	New directory for mdbserver . Contains server source code for RPC functions.
<u>debugger/mdbserver/rpc</u> :	New directory for the RPC protocol source code.
<u>debugger/backend/server</u> :	C code for the libmonodebuggerserver library. New source files remote.c and remoteinputforwarder.c .
<u>debugger/backend/arch</u> :	C code for the libmonodebuggerserver library

5. Client and server code

The following list enumerates all source code files that trigger RPCs on the host side:

- backend/server/x86-pttrace.c
- backend/server/x86-linux-pttrace.c
- backend/server/i386-arch.c
- backend/server/thread-db.c

- backend/arch/bfd/cache.c
- backend/arch/bfd/bfdio.c

The following list enumerates all source code files that execute RPCs on the target side:

- mdbserver/remotedebug_bfd.c
- mdbserver/remotedebug_server.c
- mdbserver/remotedebug_thread-db.c

6. Description of mdbserver

The **mdbserver** is a multi-threaded application. Its C **main()** function is located in the source file **mdbserver/rpc/remotedebug_svc.c**. No command line arguments are required.

The following threads are executed:

- RPC thread 1: Provides all RPC server functions except of one (see RPC thread 2). These functions are always called sequentially and hence are served by a single RPC server handle. Remote calls to these functions have the default timeout of 25 seconds.
- RPC thread 2: Provides only 1 RPC server function that executes the **waitpid()** Linux system call. The Mono debugger detects changes of the inferior status by calling this function. Because it is called concurrently with other remote functions served by RPC thread 1, an own thread and RPC server handle is required.
- Quit thread: This thread waits on a mutex which is initially locked and gets unlocked by an RPC server function. It is responsible for terminating **mdbserver** if the Mono debugger on the host terminates.

The **mdbserver** prints some information to stdout:

- Ports on which the RPC services are provided
- IP address and port of Mono debugger that connects to it
- If another Mono debugger instance wants to connect to it

7. Input to the inferior application

The Mono Debugger allows to send input to the inferior application in normal (not remote) mode on the same terminal. This is merely Linux's functionality: The application's stdin is connected to same terminal as the debugger when the debugger forks the application. Now, if the application is stopped, the input to the

terminal is grabbed by the debugger, but if the application is running, the input is directed to the application. This is the case if, for example, the method

System.Console.ReadLine()

is executed in a C# application.

To provide the same functionality for remote debugging, a tiny application named „MDB remote input forwarder“ was implemented. Its source code is located in

backend/server/remoteforwarder.c.

It is automatically compiled with the debugger and installed in Mono's **bin/** directory. It is forked locally by the debugger before the inferior is forked remotely. It is stopped and resumed together with the inferior and hence emulates it on the local terminal.

At startup, it creates a listening socket so that mdbserver on the target can make a connection to it. On the target, the socket is then connected to the inferior's stdin file descriptor.

Once the connection is established, the input forwarder application sends all the data it receives at its stdin onto the target via the socket.

8. Command-line arguments of the Mono Debugger

If debugging is done locally, the standard command line to start debugging a Mono application looks like the following:

```
mdb /home/.../mono-test/Test.exe
```

where the first argument denotes the Mono application.

For remote debugging, the command line looks like this:

```
mdb -remote-target:192.168.1.5 -remote-exe:/home/target/test/Test.exe -  
mono:/home/target/mono-bin/bin/mono /home/.../mono-test/Test.exe
```

The arguments denote the following:

- **-remote-target:** The IP address of the target where **mdbserver** executes and the inferior shall run. This is a new command line argument.
- **-remote-exe:** The absolute path of the inferior Mono application on the target. This is a new command line argument.

- `-mono`: The absolute path of the Mono binary on the target. This argument is provided by the Mono debugger to overwrite the inferior Mono.

Via these arguments, it is possible to set the paths of Mono and the inferior application separately for the target and the host. For external DLLs that were referenced at compile time it is mandatory that they live either in the Mono GAC (Global Assembly Cache) or in the same or subdirectory of the inferior application.

9. Concurrent debugging – Ports assignment

The remote debugging extension of the Mono debugger supports the concurrent remote debugging of more than one Mono application. This works as follows:

- The **mdbserver** application provides two RPC services and hence needs two listening TCP ports. It searches for two free consecutive TCP ports starting at 31000. This number is increased by two until free ports are found, where the RPC services listen for incoming requests. If no free ports can be found in the range [31000, 31098], **mdbserver** terminates with a failure message.
- The **mdbserver** application has an „occupied“ flag that is set at the first call of the RPC function **testconn()**. Other requests by calling the same function are then rejected.
- The Mono debugger tries to connect to the target RPC services starting with port 31000 at startup. If no RPC service is found or the **mdbserver** that was found is already occupied, this number is increased by two. This procedure is executed until a successful connection to an **mdbserver** was made. If no „free“ **mdbserver** can be found in the range [31000, 31098], the Mono debugger terminates with an error message.
- If the Mono debugger wants to fork the child process where the inferior executes in, it calls the corresponding RPC function remotely. To gather the inferior's output on stdout and stderr and being able to send data to the application's stdin file descriptor, three listening sockets are created on the host. Therefore, three free consecutive TCP ports starting from 32000 (range [32000, 32097]) are determined. After the sockets are bound to these port, the port numbers are sent as arguments to the **server_ptrace_spawn()** RPC.
- The socket for stdin data is created by the input forwarder application that is forked by the Mono Debugger in remote mode.
- If the **mdbserver** application executes **server_ptrace_spawn()**, it creates three sockets that connect to the host listening TCP ports. After forking the process, it redirects this process' stdin, stdout and stderr file descriptors to these sockets so that they are connected with the host and the Mono debugger.

10. Resources needed on host and target

On the host (where the Mono debugger executes), the following resources are needed:

- Mono installation.
- Mono sources from which the Mono installation was compiled. They enable the debugger to show Mono C-Code if one steps into Mono code while debugging.
- Program sources (e.g. **Test.cs**). They enable the debugger to show the C# code.
- Assemblies (e.g. **Test.exe**).
- Assembly MDB files containing symbol information (e.g. **Test.exe.mdb**). These are created by **mcs/gmcs** if the **-debug** argument is supplied.

On the target (where **mdbserver** and the inferior run), the following resources are needed:

- Mono installation (e.g. **mscorlib.dll.mdb**). The installation's version number must be identical to the one on the host.
- Assemblies (e.g. **Test.exe**).
- Assembly MDB files (e.g. **Test.exe.mdb**).

11. Installation

Follow these steps to install the Mono debugger with remote debugging capability. You need to have installed the Mono 1.2.6 framework.

- Get the debugger source code:
Go to <http://www.streamunlimited.com/vimem> and download either the patch if you want to patch your own sources, or the archive containing MDB and the remote debugging extension.
- Patch your sources (if you want to do it yourself):
Get either revision 91325 of the **debugger/** directory of the Mono SVN or download the source code from the Mono homepage (**mono-debugger-0.60.tar.bz2**).
Go to your MDB source directory and execute

```
patch -p1 < [patch file path]
```

If you downloaded the source code from the homepage, go to <http://www.streamunlimited.com/vimem> and copy the **autogen.sh** file to your source directory.

- Compile the debugger:

If you

Go to the debugger's source directory and execute

- **./autogen.sh --prefix=[Your Mono installation directory – same as above!]**
- **make**
- **make install**

- Set up the target system:

Copy the Mono installation to the target machine. Note that if its path is different on the two systems, this will not work because Mono hardcodes your path to most shell scripts located in the **bin/** directory.

If you have already installed Mono on the target machine, you may simply copy the **mdbserver** application to the target.