# BugWorld: A distributed environment for the development of control architectures in multi-agent worlds

Nikolaus Almássy
almassy@ifi.unizh.ch

Technical Report No. 93.32

Department of Computer Science, University Zurich-Irchel
Winterthurerstrasse 190, 8057 Zurich, Switzerland

August 23, 1993
Revised: December 3, 1993

**Abstract**

This software allows the simulation of several autonomous agents (simulated robots or simply *bugs*) equipped with distance sensors. Each bug can define a unique set of distance sensors located in the center of its circular body. The simulator consists of several different processes. Each new agent is a separate process that communicates with the simulator via Internet. In this way a great many of agents can be simulated at the same time on different workstations. The process that maintains all the information about the simulated world can run on a separate machine and is not slowed down by agents that might need more and more CPU time as the complexity of their architectures increases.

## Contents

# 1 Motivation

There is a growing doubt that the traditional artificial intelligence (AI) approach to intelligent behavior, that involves explicit reasoning by manipulating symbolic representations of the world, will ever lead to systems that are interacting robustly in a real world environment [Winograd and Flores, 1986, Brooks, 1991]. This raised a great deal of interest in the study of adaptive behavior and the construction of autonomous agents, that in contrast try to avoid engineered representations and relate more to the ethology of simple animals. To perform such studies one can either invest in a suitable hardware and create control architectures for real robots or, since cheap and easy usable robots are difficult to maintain, rely on robot simulations on a computer.

One might argue that simulated environments do not offer interactions that are rich enough to prove the robustness of a specific control architecture. On the other hand, testing the performance of control architectures implemented on real robots is a time-consuming task and the automation of such tests are even more difficult than building the robot itself. Hardware robots have much more complex interactions that are only possible in the real world. Hardware sensors and actuators behave in a less predictable way due to noise or hardware problems, also the morphology of the robot can influence its behavior. A control architecture that functions well in a discrete and noise-free simulation environment might fail if implemented on a robot that is equipped with noisy sensors, confronted with slippery or rough surfaces in an unpredictable environment. Such simulations of autonomous robots are then criticized for the simplicity of the possible interactions. Systematic testing of control architectures for autonomous robots is often too time consuming to rely solely on real robots. This work makes the best possible trade-off between a simple and fast simulator and a test environment that offers robot environment interactions that are rich enough such that they can be compared with real robots.

Simulations always should be combined with studies on real robots to validate the results gained by simulations. The code of the robot's control architecture just has to be modified at the interface to the sensors and actuators. This technique of integrating the presented simulator in the development of control architectures for real robots was successfully used by Mondada and Verschure, 1993.

# 2 Concept of the distributed simulator

The main idea behind the design of the simulator is the possible use of a large number of simulated vehicles (agents) in a single environment, therefore offering the possibility of interactions between agents. Such dynamic environments are computationally very expensive, therefore speed was a central issue for the design of the simulator. To ensure high performance one can either rely on SISD or SIMD super computer, or try to parallelize the application as much as possible (i.e. MIMD). As supercomputers are generally not that readily accessible, the decision was made to parallelize the simulator on a local area network of medium performance workstations. This technique proved to be a suitable way to obtain very high computing performance by Cap and Strumpen, 1993.

The simulation process is therefore separated into the simulator that computes sensory information and maintains the information about all objects known to the system, and the agent that computes the next action based upon the sensory feedback that is received from the simulator. No other information about the simulated world is available to the agent. Separate processes define the world with its obstacles and targets and display it on the screen. This design concept makes the usage of the simulator more flexible and the software easy to port to a parallel hardware (See figure 1). The simulator was implemented based on the work of Goldstein and Smith, 1991, however the code was completely reorganized, optimized and modified to make it portable to a variety of machines and operating systems. The simulator has been, and still is, successfully used in several projects [Almássy and Verschure, 1992, Verschure, 1992, Mondada and Verschure, 1993].

Other projects currently pursued are the integration of the simulation software with a parallel

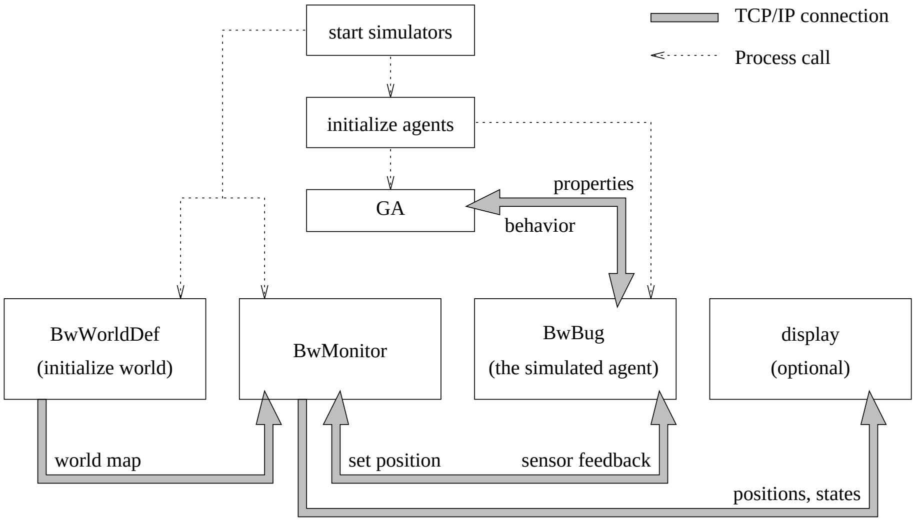


Figure 1: The Simulator *BwMonitor* is started first. Then, the *BwWorldDef* reads the world description from a file (generated by *xfig*), connects to *BwMonitor* and defines obstacles and targets. The display can connect (and disconnect) to the *BwMonitor* at any time, open up a window and constantly maintain the current state of the world with agents, targets and obstacles. Finally agents (*BwBug*) can connect to the *BwMonitor* and typically perform a sense-think-act cycle: requesting range-finder and target feedback, computing the next action based on this information, and communicating its new position to the *BwMonitor*. Once the simulation is started up it can be connected easily with a genetic algorithm (GA). The GA sends parameters to the agent in order to determine its behavioral properties. The agent reports the results (usually the number collisions and targets found) after a fixed number of steps to the genetic algorithm, which uses this information to compute the fitness.

genetic algorithm (GA). The same concepts that applied to the design of the simulator were also applied in this part: a central process manages a genetic pool of abstract descriptions of agent properties and distributes these to the agent processes. These agents are then tested in their simulation environments, and after a fixed number of steps they report the performance back to the GA process. This parallel GA is functionally independent of the robot simulation and can also be used for other GA optimizations.

# 3 Implementation

The communication between the concurrent processes is implemented by calls to the operating system routines that use the TCP/IP protocol implemented in the Berkeley socket interface [Stevens, 1990]. This communication interface is widely available on UNIX and similar operating systems. The messages are sent in ASCII format to guarantee compatibility of processes that run on machines with a different data representation. This coding and decoding slows down the communication speed, but enables the usage of a heterogeneous collection of UNIX workstations.

## Connection Establishment

In a set of communicating UNIX processes one process has to be identified as a *server*, due to the asymmetric sequence of calls of operating system routines, that are needed to initialize a connection endpoint. In our setup *BwMonitor* takes the position of the server, waits for the other processes (*clients*) to connect on a predefined port. Upon receipt of a connection, an unused port-number (given by the operating system) is sent back, on which the process has to re-connect for the actual connection. The first connection is closed again to allow other agents to connect at this port. This sub-protocol is implemented by the function *BwAttachToMonitor()*, and is not shown in Figure 5. By this method only one unique port has to be defined per machine, and collisions with other applications that use TCP/IP ports are avoided. The main improvement since previous versions of this software is that *BwMonitor* does not consume CPU time if no processes are requesting new positions or sensory information. The requests of the peers are queued and handled accordingly.

## Process Termination

Whenever one process in the setup terminates, a final disconnection message is sent to the *BwMonitor* that records this process as disconnected. Even without this final message (in case of a system failure or shutdown of a remote host) the monitor handles the broken connection appropriately and disconnects the process automatically. These mechanisms guarantee a high level of fault tolerance and are very similar to those used by Strumpen, 1993.

## Receiving the World Definition

After *BwMonitor* is started, *BwWorldDef* is used (before any simulations), and terminates after the world with its obstacles and targets is defined. This is done by sending messages to the monitor that defines objects and targets.

## Protocol

*BwMonitor* identifies the type of the client (Agent, display or *BwWorldDef*) by the first message it receives from the client (See figure 5 for a detailed description of the protocol). The main GA process is a server as well that starts up the agents (that connect to their *BwMonitor*) and waits for its clients to connect. By that only the agents that successfully connected to their monitor will be used by the GA (see also next section).

## Implementation Platforms

The software has been successfully tested on several Machines from different vendors, and operating systems (See table 1).

| Vendor | Machine | operating system | requests/sec. | time (sec.) |
|---|---|---|---|---|
| Sun | SPARC Center 2000/10 | SunOS 4.1.3 | 154.3 | 122.6 |
| HP | HP 9000/735 | HP-UX 9.01 | 90.6 | 30.0 |
| DEC | DEC alpha | DEC OSF/1 V1.2 | 90.6 | 56.5 |
| Sun | SPARCsystem-600 | SunOS 4.1.3 | 82.8 | 85.4 |
| Sun | SPARC 2 | SunOS 5.2 | 51.6 | 185.2 |
| Sun | SPARC 2 | SunOS 4.1.3 | 48.7 | 190.6 |
| IBM | RS 6000 | AIX | 25.6 | 266.6 |
| Sun | SPARC 1 | SunOS 4.1.3 | 26.2 | 373.6 |

Table 1: The performance measures denotes the number of requests (range-finder and position requests, sensor definitions) the BwMonitor could handle for the simulation of ten simple agents in the environment shown in figure 2. The fifth column gives the total CPU-time spent in system and user mode of all participating processes. The agent had 7 sensors in the range finder.

It can be compile on any BSD like UNIX operating system with a C compiler. On System V type UNIX systems the BSD compatibility libraries can be used. The display program was implemented with the aid of the *Open Windows Developer's Guide*, which makes the implementation of new features very easy. To compile it, the *XView* toolkit and libraries are needed. This toolkit is included in *Open Windows*, and the full source is available by anonymous FTP from export.lcs.mit.edu.

# 4 Genetic Algorithm connection

Another possible application of such a distributed process setup is a distributed parallel genetic algorithm (for a good introduction to genetic algorithms see [Goldberg, 1989, Davis, 1991]). Major parts of the software were re-used in the implementation of the GA. The GA process is the server that starts several clients that compute the evaluation function of the genome. Compared to parallel hardware, workstations in a local area network are not that reliable: they are sometimes turned off and on during the day, system failures or reboots can severely perturb parallel computations. Therefore robust mechanisms for fault tolerance have to be included. This was one reason to design the GA as a steady state algorithm. A generational GA would have a pool of individuals that has to be completely evaluated before a new generation can be started. Therefore re-synchronization at every generation is needed and a considerable amount of (computing) time would be lost while waiting for the slowest host to finish. A steady state GA does not need to re-synchronize except for the first generation. Thereafter all participating hosts are always performing evaluations, and load balancing can be easily introduced by setting nice priorities to the evaluating processes. The other reason for a steady state algorithm was the closer resemblance to natural evolution: no re-synchronizations or engineered mechanisms that control selective pressure and population diversity are needed. Steady state GA's can be used to speed up a robust and focused evolutionary search [Whitley, 1989].

A future report [Almássy, 1993] will document the results of several genetic algorithm experiments made with the simulator.

# 5 Usage of the simulator

## BwMonitor: *the core of the system*

The simulation begins by starting the main process which computes the sensory information. This is done with the command

```
BwMonitor [port-number]
```

A port number can be specified which will be used by all other processes for the initial connection to the monitor. If no port number is specified, the default port number 7730 will be used. This port number must not be used by other processes in the system[1], therefore another port number has to be specified for other simulations that run on the same machine. This is either done at the command line, as described above, or with an environmental variable called `BWPORTNUM`. In the *C-shell* an environmental variable is set with the command

```
setenv BWPORTNUM 7731
```

This line can also be inserted in the ~/.cshrc file that is read in as you login to the machine. The port number given in the command line takes precedence over the number set in the environment variable.

[1] Such high port numbers are rarely used by other system processes that use TCP/IP. Ask your local system manager if this port number is used by other applications, or see file /etc/services.

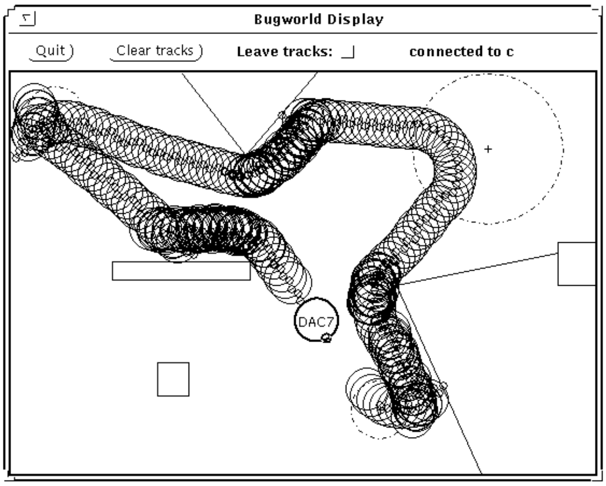


Figure 2: A simulated agent in its environment. The plus mark on either sides and the bottom of the environment represent targets that can be sensed by the agent. The circle around it determines the size of the target. The target could be "heared" anywhere in the environment, i.e. the simulator sends target information (two distances from the left and right side of the agent to the target) about all targets in the environment. All solid lines represent obstacles that are sensed by the range finder. The chain of circles is the trace of the agent that is displayed gray on a color monitor.

## BwWorldDef: *to define the world*

The world definition program is called *BwWorldDef* and is started with the command

```
BwWorldDef [host-name [environment-file [port-number]]]
```

The first argument is the host name of the machine where the *BwMonitor* is running. If no host name is specified the local host is used. The second argument is the name of the file that contains the environment specification (the default file name is `BwEnv`). The third argument overrides any setting of the environment variable `BWPORTNUM`, which specifies the port that is used by the *BwMonitor*.

The environment specification file should be generated with the public domain program *xfig*, a menu-driven tool that allows the user to draw and manipulate objects interactively in an X window[2] (see figure 3).

## display: *to see what is happening*

Now the display that shows an image of the agent's world in an X-window can be started (See figure 2):

```
display [XView toolkit options] [+/-t] host-name [port-number]
```

The XView toolkit options are described in section 3 of the UNIX manual pages and enable the selection of different colors, window sizes and the like. The panel button *Leave tracks:* can be turned off upon startup by using the option `-t` (by default the Agents will leave tracks). *BwMonitor* must run on `host-name` and use the port number `port-number`.

[2] xfig Version 2.1.7 is available via anonymous FTP on export.lcs.mit.edu in /contrib/R5fixes/xfig-patches. The output from xfig can be exported with the TransFig package (Version 2.1.7 is available on the same host) to various formats (e.g. LaTeX, Postscript, PiCTeX, ...).

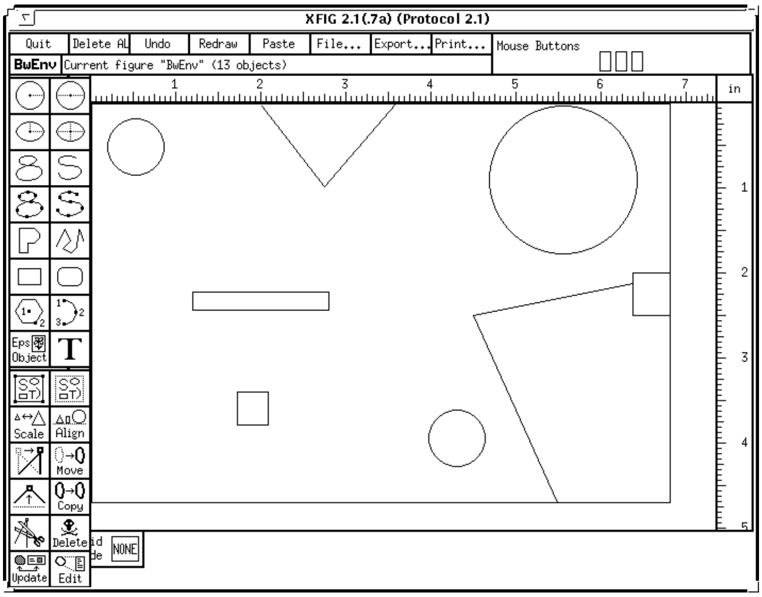


Figure 3: The program xfig is used to design an environment. The user interactively draws lines, poly-lines, boxes and polygons which represent obstacles, and circles which represent targets. No other features of the drawing program are used. The next version of the simulator will associate the fill style of a target with its "strength". Such variable targets will disappear after an agent drives over it, and reappear after some time that is determined by the fill level.

## The agent: *to test control architectures*

In the following paragraph a C - framework is given to show the required calls to connect to *BwMonitor*, to request sensor information and to send the new position to *BwMonitor*. Note that the agent is responsible for not walking through obstacles. The agents can also be implemented in C++.

```
/*
 * in this file various data structure are declared.
 * if this program is included from a C++ program, class definitions
 * will be used.
 */
#include "Bw.h"

float BwFb[MAXFB]; /* define the feedback array */
int socket_to_monitor; /* define the connection endpoint variable */
BwOBJECT bug; /* define the agent's object */
int SensorDef[BwMAXRF]; /* the description of the design of the range finder */

int main()
{
    /*
     * initialize the agent's object
     */
    bug.obj = 1; /* my number (must be unique in the simulator !) */
    bug.type = BwBUG; /* the type of the agent (always BwBUG) */
    bug.x0 = 100; /* initial position */
```

```
    bug.y0 = 100;
    bug.orientation = 90; /* measure in degrees (90 = heading south) */
    bug.x1 = 20; /* radius of the bug */
    sprintf(bug.picture, "%d", "TEST"); /* my name (appears on the display) */

    /*
     * initialize the range finder description
     */
    DefineSensor(SensorDef, 1);

    /*
     * connect to the Simulator running on host "mydec01"
     */
    if ((socket_to_monitor = BwAttachToMonitor("mydec01")) > 0)
         printf("attached to Simulator on machine %s\n", HOSTNAME);
    else error("NOT attached");

    /*
     * define the range finder on the simulator (SensorDef is an array
     * of integers that define the number of ranges and their angles)
     */
    if (!BwDefineSensors(SocketToMonitor, SensorDefinition))
         error("failed to define the rangefinder");

    for (;;) {
        /*
         * place yourself in the world (i.e. make a step)
         */
        if (!BwDefineObject(socket_to_monitor, &bug))
             error( "BwDefineObject failed" );

        /*
         * request sensor information from the BwMonitor (with target feedback)
         */
        BwGetRange(socket_to_monitor, BwFb, TRUE);

        think(BwFb, &action); /* compute the next action */
        switch (action) {
        case "F": FORWD; /* a macro to modify the coordinates of the bug-object */
                  break;
        /* and so on ... */
        }
    }
}
```

# 6 Future enhancements

## dynamic environment

It should be possible to have ways to add or delete objects or targets. At the moment the agent has to decide which target should be selected next. The Monitor should be able modify target or object positions.

- Since *BwWorldDef* can connect any time it is easy to create a dynamic environment, where new objects are introduced and others are deleted (e.g. the agent destroys an obstacle when he hits it.)
- the *BwWorldDef* should cooperate with the display, to enable the interactive maintenance of such a dynamic environment

## Flexible sensors

- Since an agent is usually just driving around (few collisions) it might be useful to change the resolution of the range finder dynamically. This again reduces the number of ranges that the *BwMonitor* has to compute and should make little difference in the resulting behavior of the agent.
- Inter-agent communication should be possible. Currently a framework for a message placement is implemented, i.e. agents can send messages like "I am in state *A*" to the monitor, that can be received by other agents upon request. Messages will be time-stamped such that they will be deleted from the monitor's memory upon timeout. By that mechanism simple rules for behavior should make it possible to generate a more complex group behavior.

## Better display

The display should offer the possibility to interactively change environments, positions and states of agents. This however must be done with an extension of the protocol currently used. For a change of the world size or obstacles currently present, the display needs to send messages to the *BwMonitor* that clears the environment, then reloads a new world definition. For state changes or changes of the location of the agents, the monitor needs to pass through the messages from the display to the agent.

- Once the *BwMonitor* terminates the display has to be restarted. A button to re-connect to a monitor on another machine might be useful.
- A closer integration of the display with the control of the world.
- The user might find scroll-bars useful if the environment size get larger than the resolution of the screen.

## Integration with a real environment

To verify the control architectures, a robot interface will be built, that allows the execution of the code of the control architectures in conjunction with a real robot with hardware sensors and actuators (see figure 4). With such an interface it is then possible to evolve control architectures for robots.

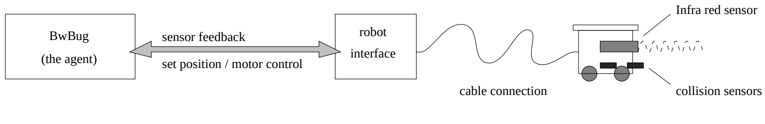


Figure 4: In a future project a robot interface will be created to better verify and test control architectures.

## 7 Conclusion

A fast and simple environment for the study of learning algorithms for autonomous agents is presented. The main point of the software is parallelism that can be exploited in a dynamic simulation environment with several workstations that run the software. As local area networks of UNIX workstations become more and more available to a larger amount of researchers, this implementation seemed the right way to grant both portability and speed. Different types of workstations can be used together because of a machine independent message format. The simulator is implemented in *C* and therefore easily portable to a large number of hardware platforms.


## Acknowledgements

I would like to thank Erik Vinkhuyzen for his inspiration and helpful comments on earlier versions of this paper, and Volker Strumpen for his support on the programming of the socket interface. Furthermore I would like to thank all colleagues who agreed to share their workstations for various CPU-time consuming experiments.


## References


[Almássy, 1993] Nikolaus Almássy (1993). A parallel steady state genetic algorithm: applications in the study of development of neural networks. Technical Report forthcoming, Department of Computer Science, University Zurich-Irchel, Zurich, Switzerland.

[Almássy and Verschure, 1992] Nikolaus Almássy and Paul Verschure (1992). Optimizing self-organizing control architectures with genetic algorithms: the interaction between natural selection and ontogenesis. In [Männer and Manderick, 1992], pages 451–460. Elsevier.

[Brooks, 1991] Rodney A. Brooks (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

[Cap and Strumpen, 1993] Clemens Cap and Volker Strumpen (1993). Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19(11):1221–1234.

[Davis, 1991] Lawrence Davis, editor (1991). *Handbook of Genetic Algorithms*, New York. Van Nostrand Reinhold.

[Goldberg, 1989] David E. Goldberg (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.

[Goldstein and Smith, 1991] Louis Goldstein and Kyle Smith (1991). Bugworld, a distributed environment for the study of multi-agent learning algorithms. University of California at Santa Cruz, Dept. of Computer Science.

[Männer and Manderick, 1992] Reinhart Männer and Bernhard Manderick, editors (1992). *Proceedings of the Second Conference on Parallel Problem Solving from Nature, Brussels, Belgium*, Amsterdam. North-Holland.

[Mondada and Verschure, 1993] Francesco Mondada and Paul Verschure (1993). Modeling system-environment interaction: The complementary roles of simulation and real world artifacts. In *Proceedings of the ECAL 93: Self organization & life: from simple rules to global complexity*, pages 808–817. Université Libre de Bruxelles.

[Schaffer, 1989] J. David Schaffer, editor (1989). *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, California. George Mason University, Morgan Kaufmann.

[Stevens, 1990] W. Richard Stevens (1990). *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, NJ.

[Strumpen, 1993] Volker Strumpen (1993). Paralle molecular sequence analysis on workstations in the internet. Technical Report 93.28, Department of Computer Science, University Zurich-Irchel.

[Verschure, 1992] Paul Verschure (1992). *Modeling adaptive behavior: the dynamics of system environment interaction.* PhD thesis, University of Zurich, Switzerland.

[Whitley, 1989] Darrel Whitley (1989). The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In [Schaffer, 1989], pages 116–121.

[Winograd and Flores, 1986] Terry Winograd and Carlos Fernando Flores (1986). *Understanding computers and cognition: a new foundation for design.* Ablex, Norwood, N.J.

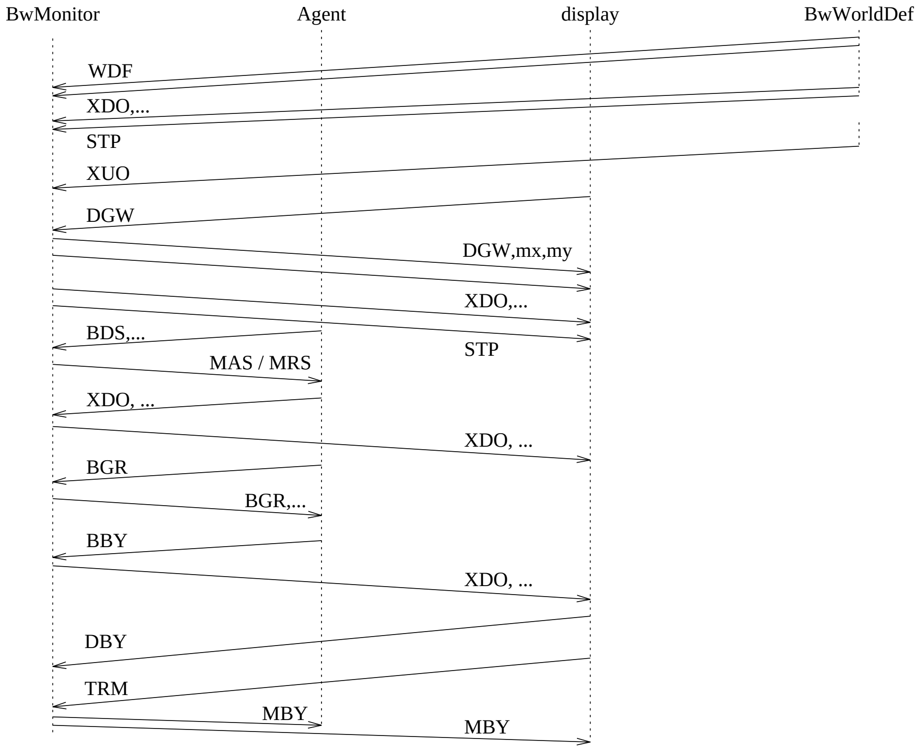


Figure 5: The BwMonitor is the server for the Agents and accepts connection request from the agent, that connect. The server identifies the client by the first message read. BwWorldDef identifies itself with the World DeFinition message: WDF, maxX, maxY, and then sends all information about defined objects with XDO, ... messages (eXtern Defined Object), concluding with a STP (SToP) message. Later, during the simulation any program (most probably another version of BwWorldDef) can send XUO (eXtern Update Object) messages to BwMonitor which will be forwarded to the display (if connected). The display first sends a DGW (Display Get World) to get the size of the world. The monitor answers with the world Size and a sequence of XDO terminated by a STP. After that the display will receive every change in the environment with a XDO message from the monitor. The Agent registers to the monitor with a BDS (Bug Define Sensor) message where the properties of the range-finder are defined. The monitor confirms this with a MAS (Monitor Accept Sensor) or MRS (Monitor Reject Sensor) depending on the validity of the definition. The agent defines its position with a XDO message that is forwarded to the display. With BGR messages (Bug Get Range), the agent requests sensory information that is answered with the same message type. Upon termination of the agent a final message BBY (Bug says good BYe) should be sent. A special XDO message will inform the display. The display sends DBY (Display says good BYe) upon termination and if the monitor crashes, terminated by a user request or receives a TRM message, a MBY message (Monitor says good BYe) will be sent to all connected clients. In this case the Agent then has to terminate, while the display remains until closed by the user.