Middleware-based distributed heterogeneous simulation

Cecil Bruce-Boye[1], Dmitry A. Kazakov[2], Helge Colmorgen[3], Rüdiger zum Beck[4], Jehan Z. Hassan[5] and Harald Wojtkowiak[6]

***Abstract*— In this paper we discuss an implementation of distributed heterogeneous simulation applications. The LabMap middleware is used as a vehicle for distribution and synchronization between simulating components. We present a simulation setup using two popular simulation tools: MATLAB/Simulink and LabVIEW. We show the feasibility and advantages of deployment of a middleware as an abstraction layer for simulation applications, in particular a smooth transition from simulation to hardware-in-the-loop setups and back.**

***Index Terms*— Middleware, Software bus, LabMap, Simulation, MatLab/SIMULINK, LabVIEW**

I. Introduction

Modern simulation applications are very complex and tend to consume a lot of resources. A lot of these applications are mutually incompatible with each other. The complexity of the simulation software requires reusing the existing simulation models without the need to translate them into another simulation language additionally even without gluing them into one model under the same tool.

Another problem of simulation applications is a transition to the real-time test environment, especially when some parts of the simulation are to be replaced by the real hardware. Due to different platforms, several time bases have to be synchronized.

A distributed middleware appears an ideal tool, on one hand, to distribute parts of simulation across multiple computing entities and on the other to abstract the differences between the simulation components and the hardware, when it comes to hardware-in-the-loop systems.

With the unified middleware layer the simulation components can be designed in the most appropriate modeling language as well as in a universal purpose programming language.

[1] **Cecil Bruce-Boye**, University of Applied Sciences Lübeck, Mönkhofer Weg 239, 23562 Luebeck, Germany
[2] **Dmitry A. Kazakov,** cbb software GmbH, Isaac-Newton-Strasse 8, 23562 Luebeck, Germany
[3] **Helge Colmorgen,** cbb software GmbH, Mittelweg 2, 38106 Brunswick, Germany
[4] **Rüdiger zum Beck,** cbb software GmbH, Isaac-Newton-Strasse 8, 23562 Lübeck, Germany
[5] **Jehan Z. Hassan,** cbb Software GmbH, Mittelweg 2, 38106 Braunschweig
[6] **Harald Wojtkowiak,** cbb Software GmbH, Mittelweg 2, 38106 Brunswick, Germany

In our earlier works [1, 2, 4] we showed how a middleware can be used for distribution of hardware-in-the-loop applications. Two novel characteristics distinguish this setup from our last ones. The first one is that the simulation runs heterogeneously; we are using two different simulation tools on different computers. The second is that the parts of a distributed application run in real time and thus necessarily stay synchronized to each other as long as they satisfy the real-time constraint. The middleware takes care of clock synchronization and data distribution.

A distributed simulation application is not synchronized this way, because the clocks used in its parts are independent, showing simulated time. There is no common reference clock. This problem of synchronization is the central issue for a distributed simulation we address in this paper.

II. Synchronization mechnisms of the middleware

The middleware LabMap provides a variety of synchronization mechanisms considered for a distributed application. All these mechanisms are bound to the states of a variable, which is the atomic object the middleware deals with. So LabMap can be considered as a message oriented middleware described in [10, 11].

A. Waiting for I/O completion

The application may enter time-limited waiting for completion of I/O involving a variable. This type of synchronization can be used with any hardware. For a distributed application the major interest naturally represents a networking hardware. A part of the application running on one host may request or send a variable to another part on another host and then block until the I/O completes. This is a one way synchronization which can be used only on the remote side. The counterpart should use another synchronization mechanism.

B. Waiting for value change

The application may enter time-limited waiting while the value of a variable is being changed. This method is very often used for monitoring system state variables. This type of synchronization between distributed parts of the application can only be used when the shared variable guarantees to change its value. A toggling bit can be used to ensure that. The synchronization is two way, if both sides write the variable. Yet it might be difficult to deploy this mechanism in a simulation language.

C. Blackboard

The application may trace all changes of a variable. For instance an application would like to trace a signal waveform for visualizing. A usual approach for catching value changes involves some kind of notification mechanism between the source of the value and the application. Such point-to-point bias is hard to implement without overstraining the system resources and a danger of running out of resources when an application ends abnormally. The software bus uses an alternative approach. All value changes are written onto the blackboard and remain there for a certain time. Any application may inspect the blackboard contents in order to trace the changes of desired variables.

D. Computable registers

Computable registers are represented in the middleware LabMap as a virtual hardware. The computable registers interface makes calculation of registers from other register values possible. It appears useful for distributed simulation applications, especially for synchronization purpose, because the language of computed registers has its own operations to evaluate the data consistency. These can be used for triggering synchronized parts of the distributed simulation. The operator ~ over two arguments yields 1 when both operands reach a non-zero value. It functions as follows: If both arguments are zero, the result is also zero. If one of the arguments becomes non-zero, the operation remembers that fact, but the result remains zero. The operator triggers only when the two arguments are both zero. Its result becomes 1, and a new cycle begins. Together with the register update operator %, this operation can be used to eliminate race condition. The expression

$$\%2810 \sim \%2820 \sim \%2830$$

yields 1 only when all three variables are updated. The updates may happen asynchronously, i.e. first 2810, then 2830 and 2820 at last. This can be used as a trigger condition for a simulation part.

III. Interfacing Simulation Software

In order to keep the simulation consistent, the simulation time of all units must run synchronous. In this work we assume that the simulation time is discrete and incremented by a constant time interval. We do not consider the cases when distributed simulations change the time constants, for example in some adaptive process, or when they have different time constants. Although this issue can be mastered by the means of computable middleware registers described above. Therefore on the fly spline interpolation operations integrated in the middleware can be considered in such cases.

The synchronization of the simulation time is achieved by blocking the simulation until a triggering event occurs. This means that the variables needed for the next simulation step are updated.

Usually simulations are mutually dependent, because the controlling loops stretch across the network, so that the same simulation is a producer and a consumer of data. Therefore a straightforward blocking until inputs change would create a deadlock. The solution in the event of change can also be triggered externally.

The triggering event in LabMap is implemented as a variable which can be changed. The simulation software interface has two triggering events: one at the beginning of the simulation step and another at its end. The step is blocked until the first event. At the end of the step, when all relevant outputs are written, the second event is triggered. The LabMap variables corresponding to the events are connected over the network.

Because LabMap has the architecture of a bus, the same step-end triggering signal can be distributed to many subscribers, i.e. routed to many step-start events. Triggering is implemented as toggling the corresponding LabMap variable between 0 and 1.

IV. The System Setup

For illustration purposes we created a distributed simulation of a control system described in [3]. A state-feedback-controller with a Luenberger observer [5, 9] is used to control speed of a DC Motor. The input to the motor model is voltage in volts and output is speed in revolution/minute. The controller and observer gains are calculated with the help of Ackermann's formula. The pre-filter is also calculated to eliminate steady-state-error. Since we used a different DC Motor from the one

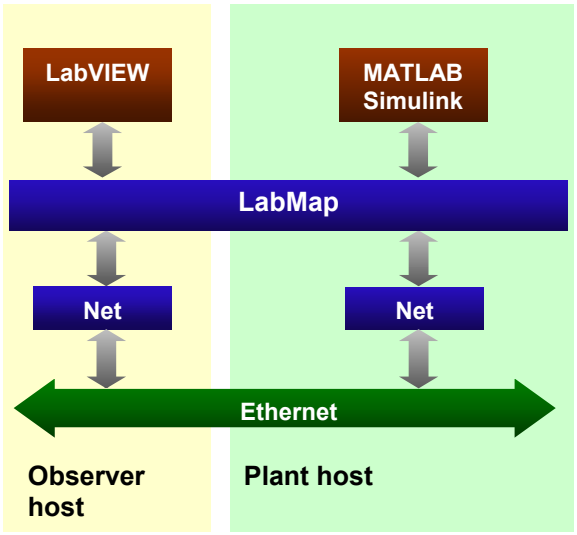

FIG.1 Distributed heterogeneous simulation

used in [3], new controller and observer gains as well as motor model parameters were calculated.

The closed loop control system model is divided into two sub-models. One model is comprised of subsystems in the forward path of closed loop system i.e. pre-filter, plant (motor model), gains etc. This model is implemented in MATLAB/Simulink [6]. The other model simulates the subsystems in feedback path of closed loop system i.e. observer and state-feedback controller. This model is implemented in LabVIEW [7]. These models run on two separate network hosts. The communication between two

models is carried out with the help of LabMap and its interfaces to MATLAB/Simulink and LabVIEW.

In the second step, the motor model in MATLAB/Simulink is replaced by a real DC Motor. The Simulink model is modified by removing the model and connecting the corresponding inputs and outputs to the motor over LabMap. In this case both models run in real time controlling the motor. We perform this second step in order to verify our simulation results.

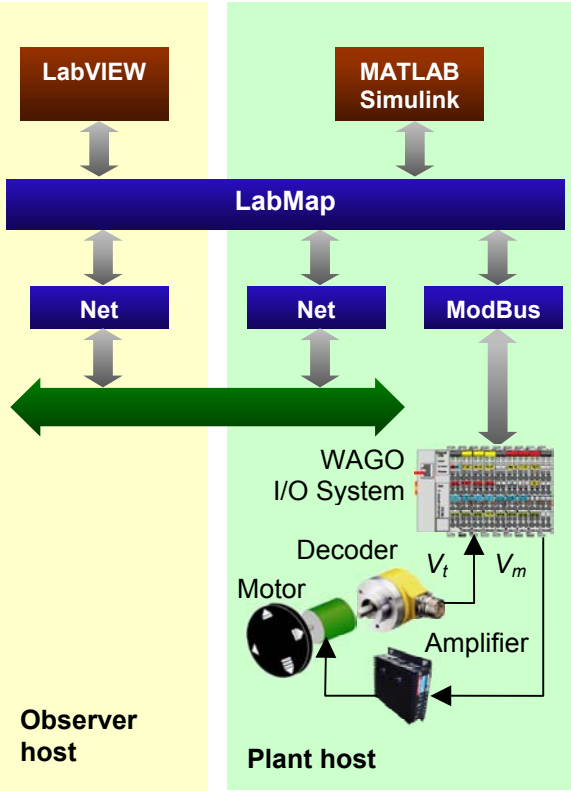

FIG.2 Real-time verification

V. Interfacing MATLAB/Simulink

The MATLAB/Simulink interface is established through a library of MATLAB s-functions, which encapsulates the standard LabMap programming interface, written with the c programming language. Fig. 3 shows an example of such a s-function for the LabMap interface. The "direction" indicates whether MATLAB should send or retrieve data from LabMap. For synchronization purposes "External Trigger" and "Completion Trigger" can be activated. It uses the value change synchronization mechanism (Chapter II). Therefore two handles "External Trigger Name" and "Completion Trigger Name" should be named. The external trigger is an input and its value will be set by 3rd party software (for instance LabVIEW). A changing value of this trigger indicates that MATLAB can now start its own calculations. After it has finished and the values were updated in LabMap, the completion trigger will change its value, and shows so, that MATLAB has finished its calculations for this step.

In case of connection or transmission errors, several kinds of behavior are possible. In this case, the whole simulation will be aborted, indicated by the value "abort" for the parameter "By Timeout". Alternatively a warning can be given. The timings for the error conditions can be set by the parameters "External Trigger Timeout" and "Completion Trigger Timeout". The complete parameter configuration can be assigned to one or more LabMap handles listened in the "HandleArray".

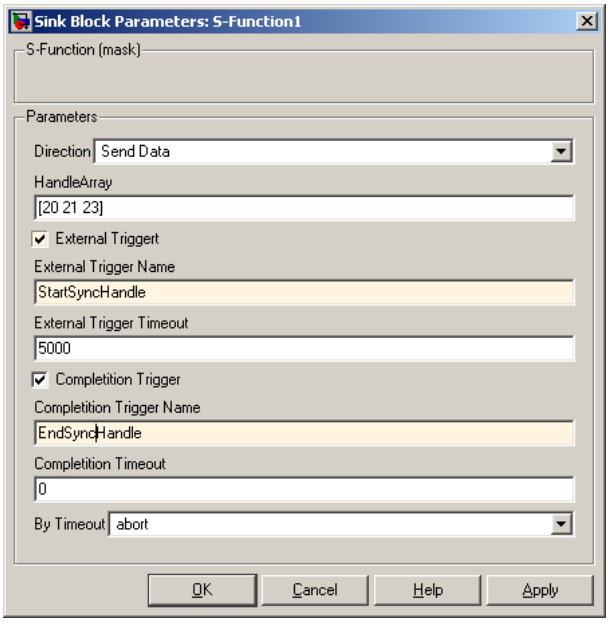

FIG.3 Parameters of the MatLab/SIMULINK interface

VI. Interfacing LabVIEW

The interface to LabVIEW is comparable to the one for MATLAB/Simulink. It also provides customized blocks for accessing process variables from the LabMap middleware in a synchronized manner.

VII. Simulation

For the test system we chose a Luenberger state observer controller with following parameters, calculated by this MATLAB script:

```
%calculation of state space controller and
%luenberger observer
%the system identificatin was carried out
%by the system identification
%software toolbox IDICON
num = [1.9618e+004] %result from system
identification
den = [1 37.7028 69.9862]  %frequency domain
Ac=[0 1; -den(3) -den(2)]; %state spaces discription
bc=[0;num];
Cc=[1 0];

km=den(3)/num              %measurment factor
cm=km*Cc;
p=[-10 -20];               %pole placement observer
gc=(acker(Ac',Cc',p))';    %state observer

prc = [-2 -2.5];           %pole placement
controller
rc=acker(Ac,bc,prc);       %state controller

vc=inv(cm*inv(bc*rc-Ac)*bc);
```

state space description of motor model:

Ac = [0 1.0000; -69.9862 -37.7028]
bc = [0; 19618]
Cc = [1 0];

prefilter and measurement factor:
vc = 0.0714
km = 0.0036

state observer and controller gains:
gc = [-7.7028; 420.4309]
rc = [-0.0033; -0.0017]

gc and *rc* are calculated by the Ackermann formula. Motor model parameters were identified with the software toolbox IDICON.

Fig. 4 represents an observer implemented in LabVIEW. The observer simulation runs on a separate computer. It communicates with the controlled system running on another host. The observer model has two synchronization blocks, which provide triggering signals from and to plant host (simulated in MATLAB/Simulink) via LabMap. Triggering the signal indicates that data is available. LabVIEW model then completes an iteration of the control loop. After the calculations have been completed the send block notifies MATLAB/Simulink and sends the data over the middleware.

The controlled system simulation is implemented in MATLAB/Simulink as shown in fig. 5. The simulation includes the motor model.

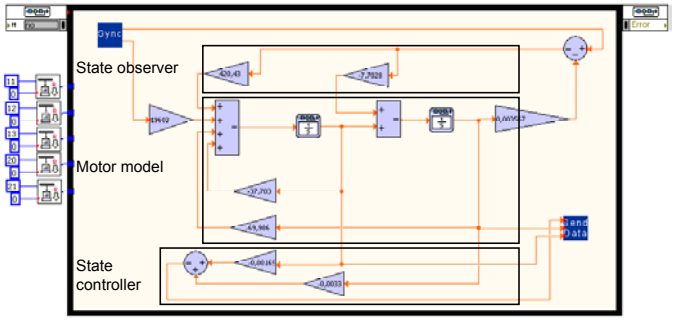

FIG.4 Simulated observer and state-feedback-controller. LabVIEW

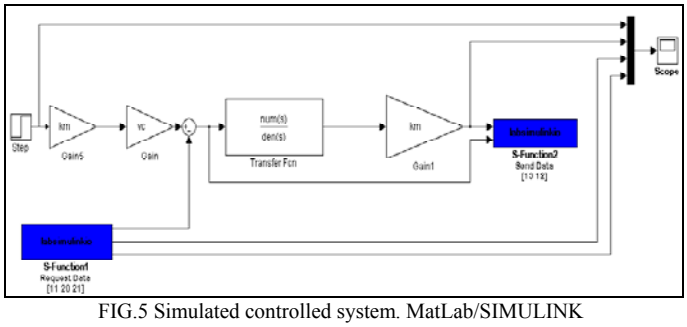

FIG.5 Simulated controlled system. MatLab/SIMULINK

In fig. 6 the following signals of the step response are plotted; set value and output value of the system, system state variables – motor velocity (state value 1) and motor acceleration (state value 2). The values of state variables are calculated by observer running on LabVIEW. It can be observed from fig. 6 that the motor velocity calculated by observer and actual motor velocity has the same response.

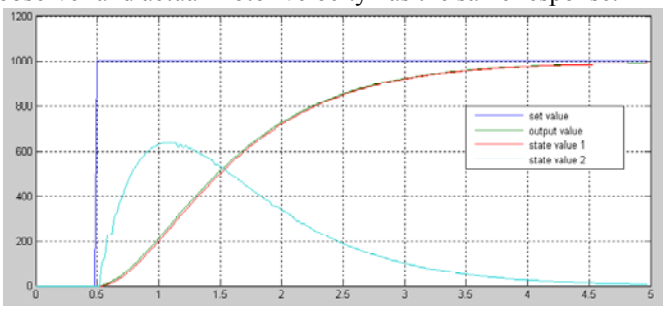

FIG.6 Simulated system response

In simulation mode the quality of data distribution services of the middleware play no significant role, because there is no real-time constraint. For the cases when the controller is to be transferred onto a target platform distributed via the middleware, the quality of service should satisfy certain constraints. In the given case using LabMap as a middleware does not deteriorate the stability of the implemented controller. The quality of data distribution services of LabMap was explored in [2]. It was shown that the latency lies between 250µs without a stress load and 1ms with such load. Because the fixed simulation time step is set to 10ms, the latency and jitter inflicted by the middleware data distribution layer should have no effect on the stability of the controller in real time.

VIII. HARDWARE-IN-THE-LOOP

In order to verify the simulation we replaced the motor model simulation block with a real motor. The observer part shown in fig 4 does not require any modification, if the system would run in real time. Necessary synchronization to real time is performed by the LabMap middleware.

In the controlled system we replaced the block simulating the motor with the blocks of LabMap communicating with the real motor over the WAGO I/O modules [8]. The modified controlled system is shown in fig 7.

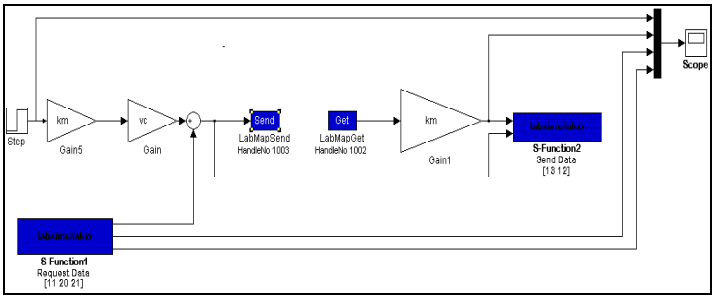

FIG.7 Controlled system modified for real motor. MatLab/SIMULINK

The system step response is shown in fig 8. It can be observed that the simulated system and system with real DC Motor produce identical transient responses.

These results verify the reliability of real-time synchronization mechanism presented in this work using LabMap.

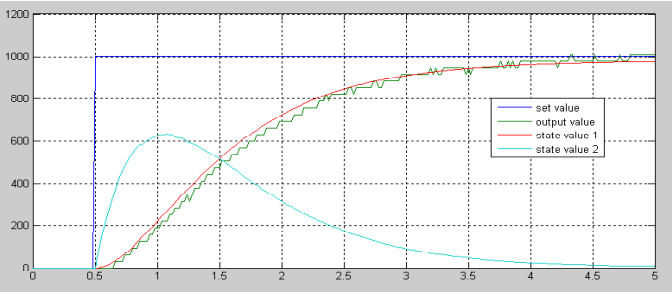

FIG.8 Response with a real DC motor

IX. CONCLUSION

The synchronization mechanisms provided by the middleware LabMap are suitable for distributed heterogeneous simulations. The middleware interface to wide spread modeling tools supports a generic synchronization of the simulation time. The designer has a choice to use it with a concrete synchronization mechanism.

The simulation model can be smoothly transformed into a hardware-in-the-loop system.

The middleware abstraction layer allows construction of heterogeneous simulations using a variety of modeling tools. The models designed in different simulation languages or tools can be reused without the need to integrate them in a unified model in one simulation language or tool. Parts of the models can be designed in general purpose languages as well.

REFERENCES

[1] C. Bruce-Boye and D. Kazakov, "Distributed data acquisition and control via software bus," Proceedings CSMITA'04, pp. 153–156, Sep 2004.

[2] C. Bruce-Boye, D. Kazakov, Quality of Uni- and Multicast Services in a Middleware. LabMap Study Case," Conference CIS^{2}E 06 (International Joint Conference on Computer, Information and System, Science and Engineering") 2006, IEEE, 4-14 December 2006

[3] C. Bruce-Boye, D. Kazakov; Rüdiger zum Beck, "An approach to distributed remote control based on middleware technology, MATLAB/Simulink-LabMap/LabNet framework", Conference CIS^{2}E (International Joint Conference on Computer, Information and System, Science and Engineering") 2005, IEEE, 10-20 December 2005.

[4] C. Bruce-Boye, D. Kazakov, "Distributed data acquisition and control via software bus", International Industrial Ethernet Development High Level Forum 2004 (IEHF 2004) in Peking, Automation Panorama No. 5

[5] D Luenberger, "An introduction to observers", IEEE Trans. Automatic Control, AC-16 1971

[6] Ashish Tewari, "Modern Control Design with MATLAB and Simulink" John Wiley and Sons, Inc. 2002.

[7] "LabVIEW Simulation Module User Manual", National Instruments, April 2004.

[8] http://www.wago.com

[9] C. Dorf, R. Bishop, „Modern Control Systems (9th Edition)", Science Press and Pearson Education North Asia Ltd. 2002, ISBN 7-03-010133-2

[10] D. Bakken, "Middleware", Washington State University

[11] G. Coulouis, J. Dollimore, T. Kindberg, "Distributed systems. Concepts and design", Addison-Wesley, fourth edition 2005, ISBN-10: 0-321-26354-5.