Quality of Uni- and Multicast Services in a Middleware. LabMap Study Case

Cecil Bruce-Boye
University of Applied Science Lübeck
3 Stephensonstrasse
Lübeck, 23562 Germany

Dmitry A. Kazakov
cbb software GmbH
1 Charlottenstrasse
Lübeck, 23560 Germany

Abstract- **The quality of service (QoS) is essential for a distributed data acquisition and control system. QoS depends on numerous factors, which are difficult to predict in advance. In this paper we present an empirical comparison of uni- and multicast based implementations of distributed middleware services. The LabMap® middleware offers both uni- and multicast data distribution services. Unicast is based on TCP, multicast on the PGM streams. We measured the performance of both transport layers on the typical 1-*n* case where multicast deployment would be possible. Our study shows that PGM is a useful complement to TCP transport, though its use should be carefully planned in advance.**

I. Introduction

A typical distributed data acquisition and control system is a loosely coupled network of nodes capable to publish and subscribe system variables. For the applications running on the nodes the network is abstracted away through the middleware. The middleware provides: naming and identity services; data distribution and transport services; information services such as browsing and time services. Quality of these services (QoS) plays a decisive role for applications to enjoy advantages of the middleware.

Within the middleware the network is abstracted as a device with some network protocol supporting the services. Thus the concrete transport protocol is abstracted as well, to allow reuse of the middleware core implementation.

In the recent past there was little choice for a middleware working over the Ethernet. It was either TCP sockets for unicast or else UDP datagrams for broadcast connections. The choice was difficult, in particular, because of unreliability and lack of traffic control of UDP. Modern multicast technology presents an answer to UDP problems. It provides efficient filtering to protect an outside network from potentially massive traffic between distinct nodes. Traffic separation is achieved physically by switches. The corresponding network protocols are available for network traffic management control from the application side. Is the multicast technology mature to meet the requirements typical for automation and control application area of middleware technology?

We carried out an extensive empirical study of QoS based on uni- and multicast transport layers on the example of the LabMap® middleware[1] [1]. One of the goals of the study was to justify empirical results of LabMap® deployment for hardware-in-the-loop scenario. [2]

II. Related Works

To the present time multicast in middleware, if implemented, was exclusively on the broadcast basis. For the CORBA (**C**ommon **O**bject **R**equest **B**rocker) [3] there exist proposals for deployment of multicast [4], but no known implementations of. For unicast services QoS measures can be found in [16].

OPC (an initiative for open data connectivity) [5] also does not provide multicast layers.

The iBus middleware [6] provides multicast services, however its transport layer is not natively multicasting and any figures about performance are unknown to us.

Seppo Sierla conducted a QoS study of NDDS (**N**etwork **D**ata **D**elivery **S**ervice) implementation of RTPS (**R**eal-time **P**ublish-**S**ubscribe) middleware interface specification for unicast services [7].

Spread is a message distribution toolkit which supports multicast messages [8]. It is not a middleware, but it can serve as a reliable multicast transport layer for a middleware. Performance data on messages services are available for spread [9].

III. Middleware architecture Overview

A. Networking

The middleware abstracts connections between nodes and represents them to the applications as publisher / subscriber relations. However, the efficiency of this abstraction highly depends on the nature of the underlying connections. In general to consider are:

- *Peer-to-peer* connections, like TCP/IP sockets. The advantage of a peer-to-peer connection is that it allows a straightforward packet filtering based on MAC (**M**edia **A**ccess **C**ontrol) addresses. An error correction mechanism is usually easy to implement, for example, by resending, because both sides are aware of each other´s state. The disadvantage of peer-to-peer connections is that in the case

[1] LabMap® is applied for testbed automation by automotive vendors like AVL, Daimler-Chrysler AG, Opel, VW, MAN, Bosch AG.

of 1-*n* and *n*-1 connections the overall overhead is a multiplicative of *n*.

- *Multicast* connections, like UDP datagram sent at the broadcast address. The advantage of multicasting lies in fixed overhead for 1-*n* connections. However using UDP might expose difficult QoS problems and middleware management problems. This problem, which is not specific to middleware, was recognized the by IETF (**I**nternet **E**ngineering **T**ask **F**orce) and a series of standards was designed to provide more efficient transport protocols suitable for 1-*n* connections. In particular IGMP (**I**nternet **G**roup **M**anagement **P**rotocol) [10] and PGM [11] (**P**ragmatic **G**eneral **M**ulticast) are of especial interest for middleware.

B. *Remote services*

The policy of QoS is the requirements imposed by a subscriber on the published data. It is the expectations of the subscriber on QoS. The following policies are important:

- *On demand* - the subscriber explicitly requires data from the publisher. This type of policy is essential for implementation of events, commands, client-server queries, higher order services such as browsing.
- *Periodic* – the subscriber receives a data flow from the publisher. The subscriber specifies the data period. This type of policy is used for physical state data known to be defined at each moment of time. Usually the subscriber asks the publisher for the "native" data period because it is the physical limit and more frequent polling makes no sense. This policy is widely used, but exposed to various problems. The period should be twice as long as the "native" period, otherwise the subscriber will experience "oversampled" data. This sufficiently limits the system performance. When timestamps are supported, the subscriber can filter out repetitive data, however this would mean an additional burden for it.
- *Periodic on change* – the subscriber receives data only upon state change. Usually the state change is value or timestamp change. This type of policy does not suffer the problems typical for the periodic policy. Yet it sometimes makes application design more difficult, because the subscriber should synchronize on the data, which depending on the data sources might stay unchanged for a long period of time. In such circumstances, for the subscriber it might become difficult to detect data losses. Another problem is that the system load is less predictable under such event-controlled scenario. For mission critical applications that could be unacceptable due to possible time constraints violation. However, in some cases time constraints can still be satisfied due to physical / logical constraints a priory known for the system.

C. *Local services*

QoS is also influenced by the middleware notification services available to the subscribers. The services can be synchronous or asynchronous to the subscriber's execution threads. For a service to be synchronous implies that the data transfer may occur only on demand and also the subscriber is blocked until I/O completion. Such architecture is obviously flawed, so in all known implementations the synchronous services are only interfaces to the underlying middleware services, which themselves are natively asynchronous. The notification services can be:

- *Callbacks* – this notification service is usually performed from a separate execution thread, which requires interlocking and data exchange with the notified thread. Further callback can be blocking or non-blocking. A blocking callback prevents data loss. That is - if a next notification needs to happen during callback processing it is postponed until callback completion. Blocking callbacks is a great danger for the whole system because they may violate time constraints, strain system resources and deadlock. Non-blocking callback may suffer data losses.
- *Synchronization objects* – this notification service is based on a waitable resource.

There are many types of synchronization objects:

- *Event* is the most simple synchronization object. An event gets signaled upon notification. The subscriber can wait for the event. This solution may also suffer data losses and data corruptions.
- *Semaphore* is a synchronization object that can be used to protect shared resources, such as data. The most used variant of semaphore is mutex. Mutex represents a very low-level mechanism exposed to various problems, from deadlock to priority inversion. Usually mutex and event are used as building blocks for higher-level synchronization objects, which are more reliable and safer to use. This notification service is based on a waitable resource.
- *Queue* is a more elaborated synchronization object. As with the callbacks the queue may be blocking or not. Thus it is again a trade-off between time constraints and data consistency. Queue represents a 1-1 synchronization object.
- *Blackboard* is a 1-*n* synchronization object. The notifications get published on the blackboard and interested threads may inspect the blackboard for the notification and enter waiting for a new notification.
- *Protected object* is a higher-level primitive, which can be specialized into each of mentioned above objects. Protected objects are language supported and can be used only in interfaces written in higher-level languages providing concurrency primitives. Protected objects are known to be very efficient in terms of context switches [12]. The disadvantage is that they require language support, which a middleware interfacing to lower-level languages like C++ would lack.

IV. MULTICAST TECHNOLOGY OVERVIEW

Two issues are essential for middleware to take an advantage from multicasting: reliability of data streams and the topology of connections, which would allow an effective mapping of publisher-subscriber relationships to the multicast groups.

Fundamentally, multicast cannot handle bidirectional connections. This excludes commands and client-side requests

from multicast. In other words, only state data can be effectively transmitted using multicast. Unicast peer-to-peer connections will be still necessary for handling some bidirectional and all feedback traffic. This includes: I/O commands, time synchronization protocol, configuration requests, browsing and informational requests. For example, it is impossible to synchronize time on the basis of a unidirectional connection.

Another requirement for multicast connections to be effective is presence of logical 1-*n* connections. Otherwise, a peer-to-peer connection is expected to be more effective. The reason for this is that in a peer-to-peer connection both sides can maintain the connection state, because the information about it is fully known to them. On the contrary, in the case of multicasting neither the producer nor its consumers know the state. This requires additional protocol overhead to make consumers able to join broadcast without producer notification. For the middleware it means a more fat protocol of sending data. In LabMap®, for instance, incremental packets are periodically intermixed with full data packets within some definite time frame. The length of the frame will determine the time a consumer will need to establish a connection to the producer.

Differently to broadcasting, multicasting was designed with reliability in mind. There are several reliable multicasting protocols. PGM is one of them supported both by the hardware and OS vendors. In particular, PGM implementations exist for Windows® and Linux. For an application, like middleware, PGM provides a reliable data stream from producer to any of the consumers. The packets used in PGM are neither TCP nor UDP. The protocol takes responsibility for resending lost packets as necessary. Technically the multicast stream is buffered in the routing nodes. The negative acknowledgements from clients propagate up to the last routing node, which has the lost packet. The packet is then resent down to the client. The routing nodes keep the traffic window to make packet resending possible. The window size and length are configurable. This integrated error correction mechanism and efficient traffic filtering based on IGMP makes PGM superior to UDP for implementation of 1-*n* connections.

The LabMap® middleware provides reliable network transport layers for unicast and multicast, called LabNet and LabPGMNet respectively. LabNet is based on TCP/IP sockets. LabPGMNet uses PGM protocol.

V. Time Synchronization

The middleware nodes are responsible for time stamping of the values of the variables and the events related to them. Time stamping requires globally applicable time stamps, which can be stored, restored and transmitted over the network. This inevitably requires applying UTC (**U**niversal **C**oordinated **T**ime) [13] timestamps rather than local political time.

The absolute accuracy of the time stamps is usually not required to be very exact. Normally, ±1.0s would be suitable for all purposes. At the same time the relative accuracy within the distributed system need to be much better. The relative clock readings on different nodes have to be far under the 1ms accuracy margin.

There are two scenarios of synchronizing timestamps in the system:

- Synchronization of the time sources. For example, by using synchronized atomic clocks, or by distributing time signals of the same clock among all participants.
- Translation of the time stamps to one base.

The first approach is more universal but also more expensive and fragile. It also presents a difficult maintenance problem by requiring an access to all participants. With the second approach the clocks of data sources remain intact, but the time stamps are adjusted as they arrive at the subscriber side.

LabMap® offers both options. We used timestamps translation in our experiments, because it is the most probable scenario.

VI. Quality of Service

QoS is characterized by:

- *Latency*, the time required to deliver the data from one the publisher to subscriber. It is composed out of many components as shown on fig. 1. The total latency experienced by the observer is t_5-t_1. On fig. 1 t_1 is the issue time of a state change. The network interface becomes aware of the state change at t_2. The change in the form of network packets arrives at the remote host at t_3. The middleware there gets notified at t_4. And finally, at t_5 the observer receives a notification.

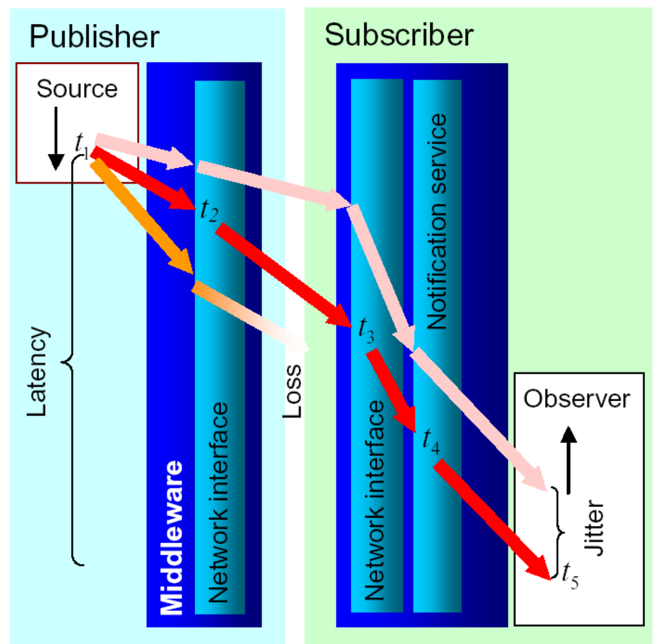

Fig. 1. Quality of service

- *Jitter*, the variation from one period to the next adjacent period of the periodic data delivery. For non-periodic data, jitter is variance of latency from its expected value. In some cases jitter is irrelevant provided the overall latency does not exceed some upper limit.

- *Data loss*, non-delivery. Data loss might be acceptable under some circumstances for periodic data.

VII. Testbed and Procedure

The testbed network was comprised out of 6 identical computers running Microsoft® Windows® Server 2003 Standard Edition:

- Intel® Pentium® 4 Processor 519 (3.06 GHz);
- Mainboard with FSB 533 MHz;
- 750 MB DDR SDRAM PC400;
- 80 GB HDD;
- NET- 3COM 3C2000, gigabit network card;
- The computers were interconnected using CISCO Catalyst 3560 switch.

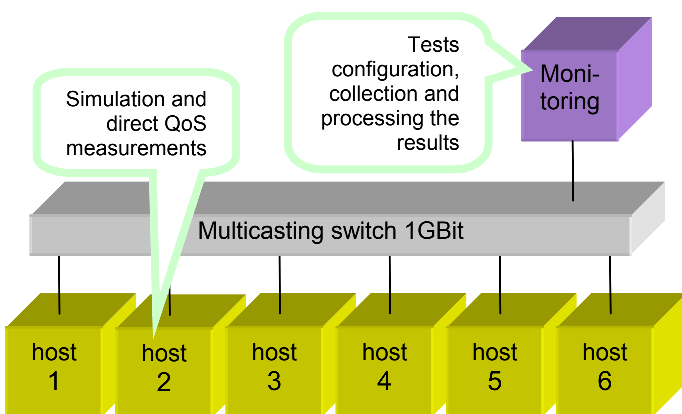

Fig. 2. Simulation configuration

Fig. 1 illustrates the simulation configuration. The hosts 1..6 simulated the nodes of a distributed control system connected via middleware. Each host ran the LabMap® middleware and a test application simulating data distribution and estimating QoS. The number of distributed variables was varied n=1, 2, 5, 10,..., 20,000 per node. We used floating-point variables in our experiments. The number of nodes s was varied s=2, 3, 4, 5, 6. Each test was executed once using the unicast transport layer and once using the multicast one. During the test each publisher application created n-variables, each of them was subscribed by s-subscriber nodes. The publishing period, i.e. the frequency in which the variables were changed on the publisher side was a parameter of the test. We used the periods Δt=10, 50, 100, 200ms. Further, each test pass was repeated multiple times (usually 1000) and the results of the repetitions were averaged.

The subscribers measured QoS directly by subtracting the notification time from the timestamp of the published variable value. Jitter was estimated as the standard deviation (σ) of the latency.

Data loss was controlled by using a definite pattern of published values. However, this was rather a plausibility check, because a total loss was not possible. LabMap® warranties delivery through degrading QoS, as long as the connection stays. For this reason we redefined data loss as a time constraint violation, i.e. as an inability for the publisher to update a variable within its publishing period. When the publisher discovered that it published only p of n variables in Δt, then n-p data losses were signaled.

All tests were performed in two variants:

- Without additional system load. The nodes carried out only simulation of variables state changes, data distribution and QoS measures;
- Under stress load, when each node ran a numerical application.

For the load simulation we chose a public-domain implementation of Whetstone [14] by Painter Engineering, Inc [15]. It was modified in two aspects. The main program was changed to a subprogram wrapped by a C++ class derived from a task type. The object of the class started a thread, which ran in parallel to the QoS test's threads. This thread performed the benchmark. Another change was that the benchmark loop did not stop. The number of cycles passed were requested from outside. When a QoS test started the current counter of Whetstone cycles was queried and the time was noted. When a QoS test stopped the new counter was taken and the difference of the counters divided to the difference of the times gave the benchmark value.

The QoS test thread and the middleware process were given higher priority to one of the benchmark thread, in order to ensure service. So the benchmark measured during the test indicated free computation resources left.

VIII. Time Synchronization Measurements

Time synchronization measures were essential, because quality of time synchronization determined the accuracy of all further measurements. The major factor was the roundtrip time of the synchronization packets sent between the hosts. Fig. 3 represents typical roundtrip times experienced for synchronization packets sent each 100ms.

The mean value of the roundtrip was about 200µs. Variance of the roundtrip is addressed to non real-time behavior of the system as a whole.

The time synchronization mechanism of LabMap® performs statistically. The estimation algorithm weights data samples according to the roundtrip time experienced. Thus samples having longer roundtrip have lesser influence on the outcome than the values with shorter roundtrips. Further, the samples expire with time, i.e. the age of a sample reduces its weight. The reason for this is that clocks of the nodes have constant drift.

A half of the average roundtrip time should roughly correspond to the upper bound of time error.

An unexpected synchronization problem we stated in our experiments was rather poor quality of the hardware clocks. An

estimated clock shift between two hosts had a constant trend. The trend depended on the computers and was up to 0.7µs/s. All computers used in the test were of the same model from the same manufacturer.

IX. Quality of Service Measurements

To reduce the dimensionality of the result space we combined the number of variables n, and the publishing period Δt into an integral parameter $n/\Delta t$, which characterize the number of middleware variables state changes per second. We discovered that the value of $n/\Delta t$ is a key factor that influences QoS. Within wide bounds, the number of variables can be safely increased when the publishing period is prolonged and reverse.

A. Losses

Fig. 4 and 5 represent losses by uni- and multicast correspondingly. As expected, multicast data distribution does not depend on the number of subscribers. The performance of

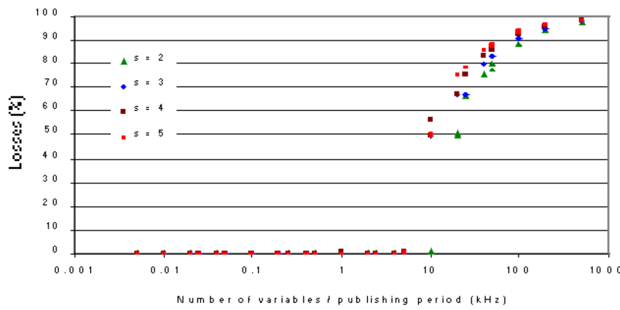

Fig. 4. Unicast losses under stress load¶

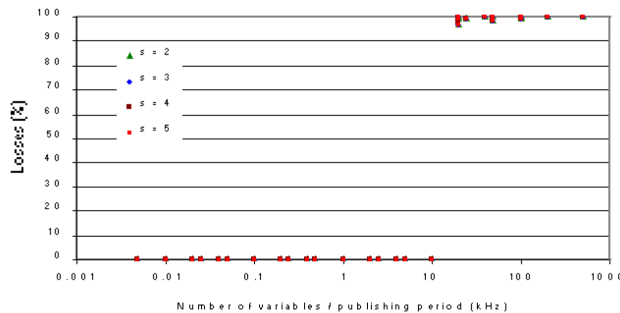

Fig.5. Multicast losses under stress load¶

unicast distribution degrades with the number of subscribers. The maximal changes frequency in both cases was about 10^4 state changes per second. This practically means, that for example 500 variables could be published no more frequently than in 50ms period.

An important observation about multicast distribution was an abrupt rise of losses in the transition area. It means that a multicast-based system should be planned more thoroughly to have more spare resources than a unicast-based system.

B. Latency

We refined latency measurement results to only the data for which no losses were observed. Otherwise, a non-delivery could eventually improve estimated latency results due to lesser data traffic, because the publisher discarded the changes it was unable to publish in time.

Table 1 represents the measured latency regression coefficients estimated for the model:

$$T_l(n, \Delta t) = a_l + b_l \frac{n}{\Delta t} \qquad (1)$$

In (1) T_l is the latency as a function n (number of variables) and Δt (publishing period). The coefficients a_l and b_l describe latency as a linear function of.

The table represents the figures obtained with a stress load (outside brackets) and ones without load (in brackets) for different numbers of subscribers s. The coefficient a_l gives an impression of the best achievable performance. Without stress load it is about 250µs for both transport layers. The coefficient b_l determines how latency grows with the growth of the number of state changes per second. The figure 10^4 (state changes per second) determines the longest latencies to observe immediately before data loss would appear. Under stress, for unicast it is around 6ms, while for multicast it is 50ms. Without a stress load they are 1 and 25ms correspondingly. Here again, multicast shows to be more fragile than unicast.

TABLE 1
LATENCY REGRESSION COEFFICIENTS

s	a_l (ms)		b_l (10^{-3})	
	unicast	multicast	unicast	multicast
1	0.998 (0.265)	0.244 (0.242)	0.521 (0.0526)	5.297 (2.349)
2	0.619 (0.151)	0.388 (0.230)	0.303 (0.0194)	5.248 (2.353)
3	0.544 (0.237)	0.246 (0.227)	0.223 (0.0866)	5.313 (2.352)
4	0.592 (0.185)	0.254 (0.213)	0.205 (0.0241)	5.284 (2.357)
5	0.710 (0.197)	0.211 (0.230)	0.178 (0.0233)	5.311 (2.361)

Unicast behavior clearly depends on the number of subscribers. b_l depends on s^{-1} almost linearly.

The results show an unexpectedly high influence of stress load, which indicates that the operating system resource sharing could be better. We expect that the performance under stress load could drastically improve on a dual-core machine.

C. Jitter

Fig. 6 and 7 represent jitter measurements under stress load. The jitter was estimated as the standard deviation of latency. As for the latency the results were refined from losses. We observed growth of jitter correlated with the growth of latencies.

Without stress load jitter was sufficiently lower. It was less than 1ms for unicast and less than 6ms for multicast.

Higher values under stress can be addressed to poor time sharing in the system.

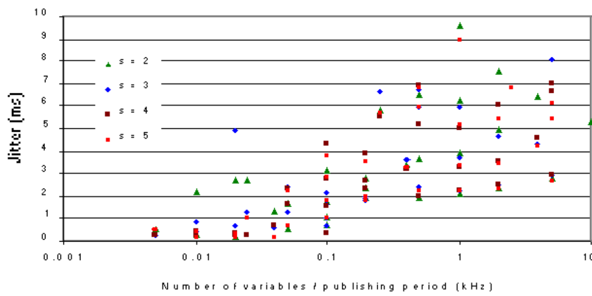

Fig. 6. Unicast jitter under stress load

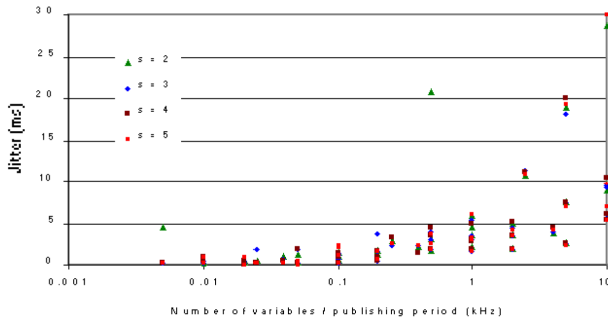

Fig. 7. Multicast jitter under stress load

D. Resources consumption

During the tests, we measured the stress load benchmarks to determine how much computational resources were left for the application logic while running the middleware. Table 2 represents the regression coefficients estimated for the model:

$$W(n, \Delta t) = a_w + b_w \frac{n}{\Delta t} \quad (2)$$

In (2) W is the number of whetstones per second as a linear function of the relation n (number of variables) to Δt (publishing period). The coefficients a_w and b_w, were estimated on both publisher and subscriber sides.

TABLE 2
WHETSTONE BENCHMARK REGRESSION COEFFICIENTS

s	a_w (10^6 whetstones · s^{-1})				b_w (10^3 whetstones)			
	unicast		multicast		unicast		multicast	
	Publ.	Subsc.	Publ.	Subsc.	Publ.	Subsc.	Publ.	Subs.
1	0.735	0.742	0.740	0.743	-0.036	-0.047	-0.033	-0.059
2	0.741	0.739	0.739	0.743	-0.061	-0.048	-0.033	-0.059
3	0.741	0.741	0.739	0.743	-0.080	-0.053	-0.033	-0.059
4	0.740	0.741	0.738	0.743	-0.097	-0.053	-0.033	-0.059
5	0.738	0.741	0.738	0.743	-0.117	-0.055	-0.033	-0.059

As expected, there is no dependency on the number of subscribers for the unicast subscriber side and multicast in general. For unicast publisher benchmarks linearly depend on the number of subscribers. Performance of unicast and multicast is comparable in other respects. Unicast subscribers slightly outperform multicast ones. On the publisher side multicast pays off already with just two subscribers.

X. CONCLUSION

Modern multicast technology provides a viable alternative to traditional UDP-based approach to implementation of 1-n logical connections. The PGM multicast protocol is reliable and imposes an overhead and QoS comparable with TCP-based connections. In general for both TCP and PGM reliable data streams, latencies in order of 250µs are achievable. This opens a wide prospective for deployment of the middleware technology in the real-time and embedded application areas with very tight control cycles.

The Windows® operating system can be deployed for less demanding distributed control applications. The periods of 5ms are realistic and reliable. However, under a stress load the system may expose latencies up to 60ms.

For process automation the LabMap® middleware provides an attractive option of utilizing multicast technology with sufficient reducing network traffic and CPU load in the nodes. The middleware and network technologies are mature for developing software products analogous to LabMap® for real-time and embedded platforms.

REFERENCES

[1] LabMap Handbook, http://www.cbb-software.com/labmap.html

[2] C. Bruce-Boye, D. A. Kazakov and R. zum Beck, “An approach to distributed remote control based on middleware technology, MATLAB/Simulink - LabMap/LabNet framework”, International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering CIS²E 05, 2005

[3] *The Common Object Request Brocker: Architecture and Specification*, OMG Document 99-10

[4] João Orvalho and Fernando Boavida, “Augmented Reliable Multicast CORBA Event Service (ARMS): A QoS-Adaptive Middleware,” Interactive Distributed Multimedia Systems and Telecommunication Services: 7th International Workshop, IDMS 2000, Enschede, The Netherlands, October 2000. Proceedings

[5] F. Iwanitz, J. Lange "OPC - Fundamentals, Implementation and Application", 2002

[6] *Professional Java Mobile Programming*, Publisher: Wrox Press Ltd. ISBN: 1861003897

[7] Seppo Sierla, “Middleware solutions for automation applications - case RTPS,” Helsinki University of Technology Information and Computer Systems in Automation Espoo 2003 Report 9

[8] Yair Amir and Jonathan Stanton, "The Spread Wide Area Group Communication System," Technical Report CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University.

[9] Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz and Jonathan Stanton, “The Spread Toolkit: Architecture and Performance,” Technical Report CNDS-2004-1

[10] *RFC 2236 - Internet Group Management Protocol*, Version 2, 1997

[11] *RFC 3208 - PGM Reliable Transport Protocol Specification*, 2001

[12] *Ada 95 Rationale: The Language, The Standard Libraries*. John Barnes (ed.), Lecture Notes in Computer Science, vol 1247. Springer-Verlag, 1997, ISBN 3-540-63143-7

[13] *ITU-R Recommendation TF.460-4: Standard-frequency and time-signal emissions*, International Telecommunication Union.

[14] H. J. Curnow and B. A. Wichman, “A Synthetic Benchmark,” Computer Journal 19 (1), February, 1976.

[15] “C Converted Whetstone Double Precision Benchmark, Version 1.2, (22 March 1998),” Painter Engineering, Inc., http://www.netlib.org/benchmark/whetstone.c

[16] Douglas C. Schmidt and Carlos O'Ryan “Patterns and Performance of Distributed Real-time and Embedded Publisher/Subscriber Architectures," Journal of Systems and Software, 2002.