How the Hidden Hand Shapes the Market for Software Reliability

Ken Birman, Coimbatore Chandersekaran, Danny Dolev, Robbert van Renesse

***Abstract*— Since the 18th century, economists have recognized that absent government intervention, market forces determine the pricing and ultimate fate of technologies. Our contention is that the "hidden hand" explains a series of market failures impacting products in the field of software reliability. If reliability solutions are to reach mainstream developers, greater attention must be paid to market economics and drivers.**

I. Introduction

We suggest in this paper that reliability issues endemic in modern distributed systems are as much a sign of market failures as of product deficiencies or vendor negligence. With this in mind, we examine the role of markets in determining the fate of technologies, focusing on cases that highlight the interplay between the hidden hand of the market (c.f. Adam Smith) and consumer availability of products targeting software reliability.

The classic example of a market failure involved the early development of client-server computing products. Sparked by research at Xerox PARC that explored the basic issues in remote procedure call, industry embraced the client-server paradigm only to discover that the technology was incomplete, lacking standards for all sorts of basic functionality and tools for application development, debugging and operational management. Costs were very high and the market stumbled badly; it didn't recover until almost a decade later, with the emergence of the DCE and CORBA standards and associated platforms.

Our market-oriented perspective leads to a number of recommendations. Key is that work intended to influence industry may need many steps beyond traditional academic validation (even when evaluation is exceptionally rigorous). To have impact, researchers need to demonstrate solutions in the context of realistic platforms and explore the costs and benefits of their proposals in terms of the economics of adoption by vendors and by end-users. Skipping such steps often inhibits promising technologies from reaching commercial practice. On the other hand, we also believe that not all work should be judged by commercial value!

Birman and Van Renesse are with the Department of Computer Science, Cornell Univerisity, Ithaca NY 14850; Dolev is at Cornell on leave from the Hebrew University, Jerusalem, Israel.; Chandersekaran is at IDA in Washington, DC. Emails {ken,rvr,dolev}@cs.cornell.edu; cchander@ida.org. This work was supported in part by grants from NSF, DARPA, AFRL, AFOSR and Intel.

II. The Hand of the Market

The distributed systems reliability community has offered a tremendous range of solutions to real-world reliability problems over the past three decades. Examples include:

- *Transactions and related atomicity mechanisms*, for databases and other services [7][4].
- *Reliable multicast*, in support of publish-subscribe, virtually synchronous or state-machine replication, or other forms of information dissemination [1].
- *The theory of distributed computing* [3][6].

Our own research and industry experiences have touched on all of these topics. We've derived fundamental results, implemented software prototypes, and led commercial undertakings. These experiences lead to a non-trivial insight: market acceptance of reliability technology has something to do with the technology, but far more to do with:

- *Impact on the total cost of building, deploying and operating "whole story" solutions.*
- *Credibility of the long-term vision and process.*
- *Compatibility with standard practice.*

Whether or not a project is attentive to these considerations, market forces often decide the fate of new ideas and systems. By understanding factors that influence this hidden hand, we can be more effective researchers.

Of course, not all research projects need to achieve commercial impact. Thus we're driven to two kinds of recommendations: some aimed at work that will be measured by its direct impact on the marketplace; others aimed at work seeking theoretical insights, where direct commercial impact isn't a primary goal. In the remainder of this document, we flesh out these observations, illustrating them with examples drawn from the major technology areas cited earlier.

III. A Brief History of Reliable Computing

In this section, we discuss the three major technology areas listed earlier. Our focus is on scenarios where the hidden hand played a detectable role.

A. Transactions and related technologies.

The early days of data storage systems were characterized by the exploration of a great range of design paradigms and methodologies. Two major innovations eventually swept most other work to the side: relational databases and the relational query algebra, and the transactional computing

model. We'll focus on the latter [7][4].

Transactions were a response to conflicting tensions. Database platforms need high levels of concurrency for reasons of performance, implying that operations on servers must be interleaved. Yet programmers find it very difficult to write correct concurrent code. Transactions and other notions of atomicity offered a solution to the developer: under the (non-trivial!) assumption that data is stored in persistent objects identifiable to the system (database records, Java beans, etc), transactions offer a way to write programs "as if" each application runs on an idle system. Transactions also offer a speculation mechanism, through the option of aborting a partially executed transaction. Finally, they use abort as a simple and powerful fault-tolerance solution.

Transactions have been a phenomenal market success and are a dominant programming model for applications where the separation of data and code is practical. However, not all programs admit the necessary code-data separation. Overheads are sometimes high, and scalability of transactional servers has been a challenge. Thus attempts to import the transactional model into a broader range of distributed computing settings, mostly in the 1980's, had limited success. Users rejected the constrained world this demanded.

To highlight one case in which a market failure seems to have impacted the transactional community: transactions that operate on multiple objects in distinct subsystems give rise to what is called the nested transaction model, and typically must terminate with a costly two or three-phase commit. Most distributed systems require this mechanism, and it is also used when replicating transactional servers. Multi-phase commit is well understood and, at least for a while, was widely available in commercial products. But products using these features were often balky, unwieldy and costly. Today, popular standards such as Web Services transactions permit the use of complex nested transactions, but few products do so. Most only offer singleton atomic actions side by side with other options (such as "business transactions", a form of scripting tool). The market, in effect, rejected the casual use of transactions that access multiple servers.

In the next two sections, we'll look more closely at reliability options for applications that have traditionally been unable to use transactions. Examples include time-critical services, lightweight applications with in-memory data structures and distributed programs that cooperate directly through message passing (as opposed to doing so indirectly, through a shared database).

B. Reliable multicast.

Three of the authors were contributors in the field of reliable multicast technologies. For consistency, we'll summarize this work in light of the transactional model. Imagine a system in which groups of processes collaborate to replicate data, which they update by means of messages multicast to the members of the group (typically, either to *all operational members*, or to a *quorum*). Such a multicast must read the group membership (the current *view*), then deliver a copy of the update to the members in accordance with desired ordering. We can visualize this as a lightweight transaction in which the view of the group is first read and data at the relevant members is then written. Joins and leaves are a form of transaction on the group view, and reads either access local data or fetch data from a suitable quorum.

In our past work, we invented and elaborated a group computing model called *virtual synchrony* [1] that formalizes the style of computing just summarized. The term refers to the fact that, like the transactional model, there is a simple and elegant way to conceive of a virtually synchronous execution: it seems as if all group members see the group state evolve through an identical sequence of events: membership changes, updates to replicated data, failures, etc. The execution "looks" synchronous, much as a transactional execution "looks" serial. On the other hand, a form of very high-speed ordered multicast can be used for updates, and this can sometimes interleave the delivery of messages much as a transactional execution can sometimes interleave the execution of operations.

Virtual synchrony has been a moderate success in the harsh world of commercial markets for reliability. One of the authors (Birman) founded a company, and virtual synchrony software it marketed can still be found in settings such as the New York Stock Exchange and the Swiss Exchange, the French Air Traffic Control System, and the US Navy's AEGIS warship. Other virtual synchrony solutions that we helped design and develop are used to provide session-state fault-tolerance in IBM's flagship Websphere product, and in Microsoft's Windows Clustering product. We are aware of two end-user development platforms that currently employ this model, and the CORBA fault-tolerance standard uses a subset of it.

The strength of virtual synchrony isn't so much that it offers a strong model, but that it makes it relatively easy to replicate data in applications that don't match the transactional model. One can, for example, replicate the data associated with an air traffic control sector, with extremely strong guarantees that updates will be applied in a consistent manner, faults will be reported and reacted to in a coordinated way, and the system will achieve extremely high levels of availability. These guarantees can be reduced to mathematics and one can go even further with the help of theorem proving tools, by constructing rigorous proofs that protocols as coded correctly implement the model.

Yet virtual synchrony was never a market success in the sense of transactions. End users who wish to use virtual synchrony solutions have few options today: while the technology is used in some settings and hidden inside some major products, very few end-user products offer this model and those products have not been widely adopted.

Virtual synchrony has a sibling that suffered a similar market failure. Shortly after virtual synchrony was introduced, Leslie Lamport proposed Paxos, a practical implementation of his *state machine model*, which also guarantees that processes in a system will see identical events

in an identical order. Over time, he implemented and optimized the Paxos system, and it was used in some research projects, for example to support a file system.

Paxos is closely related to virtual synchrony: there is a rough equivalence between the protocols used to implement group views (particularly for networks that experience *partitioning*) and the Paxos data replication model. Paxos updates guarantee stronger reliability properties than virtual synchrony multicast normally provides, but at a cost: Paxos is slower and scales poorly relative to most virtual synchrony implementations. Like virtual synchrony, Paxos never achieved broad commercial impact.

Thus, we see reliable multicast as a field in which a tremendous amount is known, and has been reduced to high quality products more than once, yet that seems plagued by market failure. We'll offer some thoughts about why this proved to be the case in Section IV.

C. Theory of distributed computing

Finally, consider the broad area of theoretical work on distributed computing. The area has spawned literally thousands of articles and papers, including many fundamental results widely cited as classics. Despite its successes, the theory of distributed computing has had surprisingly little direct impact. Unlike most branches of applied math, where one often sees mathematical treatments motivated by practical problems, the theory of distributed computing has often seemed almost disconnected from the problems that inspired the research. Moreover, while applied mathematics is often useful in practical settings, distributed systems theory is widely ignored by practitioners. Why is this? Two cases may be helpful in answering this question.

1) Asynchronous model.

The asynchronous computing model was introduced as a way of describing as simple a communicating system as possible, with the goal of offering practical solutions that could be reasoned about in a simple context but ported into real systems. In this model, processes communicate with message passing, but there is no notion of time, or of timeout. A message from an operational source to an operational destination will eventually be delivered, but there is no sense in which we can say that this occurs "promptly". Failed processes halt silently.

The asynchronous model yielded practical techniques for solving such problems as tracking potential causality, detecting deadlock or other stable system properties, coordinating the creation of checkpoints to avoid cascaded rollback, and so forth. One sometimes sees real-world systems that use these techniques, hence the approach is of some practical value.

But the greatest success of the asynchronous model is also associated with a profound market failure. We refer to the body of work on the *asynchronous consensus* problem, a foundational result with implications for a wide range of questions that involve agreement upon a property in an asynchronous system. In a seminal result, it was shown that asynchronous consensus is impossible in the presence of faults unless a system can accurately distinguish real crashes from transient network disconnections [3].

Before saying more, we should clarify the nature of the term "impossible" as used above. Whereas most practitioners would say that a problem is impossible if it can never be solved (for example, "it is impossible to drive my car from Ithaca to Montreal on a single tank of gas"), this is not the definition used by the consensus community. They take *impossible* to mean "can't always be solved" and what was actually demonstrated is that certain patterns of message delay, if perfectly correlated to the actions of processes running an agreement protocol, can indefinitely delay those processes from reaching agreement. For example, a fault-tolerant leader election protocol might be tricked into never quite electing a leader by an endless succession of temporary network disconnections that mimic crashes. As a practical matter, these patterns of message delay are extremely improbable – less likely, for example, than software bugs, hardware faults, power outages, etc. Yet they are central to establishing the impossibility of guaranteeing that consensus will be reached in bounded time.

FLP is of tremendous importance to the theoretical community. Indeed, the result is arguably the most profound discovery to date in this area. Yet the impossibility result has been a source of confusion among practitioners, particularly those with just a dash of exposure to the formal side of distributed computing, a topic explored in [5]. Many understand it as claiming that it is impossible to build a computer system robust against failures – an obvious absurdity, because they build such systems all the time! For example, it is easy to solve consensus as an algorithm expressed over reliable multicast. In effect, a multicast platform can be asked to solve an impossible problem! Of course, the correct interpretation is a different matter; just as the systems they build can't survive the kinds of real-world problems listed above, FLP simply tells us that they can't overcome certain unlikely delay patterns. But few grasp this subtlety.

This isn't the only cause for confusion. One can question the very premise of proving the impossibility of something in a model that is, after all, oversimplified. In real systems, we have all sorts of "power" denied by the asynchronous model. We can talk about probabilities for many kinds of events, can often predict communication latencies (to a degree), and can exploit communication primitives such as hardware-supported multicast, cryptography, and so forth. Things that are impossible without such options are sometimes possible once they are available, and yet the theory community has often skipped the step of exploring such possibilities.

2) Synchronous model.

The confusion extends to a second widely used theoretical model: the *synchronous model*, in which we strengthen the communications model in unrealistic ways. This is a model in which the entire system executes in rounds, with all messages sent by correct processes in a round received by all other

correct processes at the outset of the next round, clocks are perfectly synchronized, and crashes are easily detectable. On the other hand the failure model includes *Byzantine failures*, namely cases in which some number of processes fail not by crashing, but rather by malfunctioning in arbitrary, malicious and coordinated ways that presume perfect knowledge of the overall system state.

A substantial body of theory exists for these kinds of systems, including some impossibility results and some algorithms. But even for problems where the Byzantine model makes sense, the unrealistic aspects of the synchronous model prevent users from applying these results directly. For many years, Byzantine Agreement was therefore of purely theoretical interest: a fascinating mathematical result.

Byzantine Agreement has experienced a revival illustrative of the central thesis of this paper. Recently, Castro and Liskov [2] and others reformulated the Byzantine model in realistic network settings, then solved the problem to build ultra-defensible servers that can tolerate not just attacks on subsets of members, but even the compromise (e.g. by a virus or intruder) of some subset of their component processes. Given the prevalence of viruses and spyware, this new approach to Byzantine Agreement is finding some commercial interest, albeit in a small market. In effect, by revisiting the problem in a more realistic context, this research has established its practical value.

IV. DISCUSSION

We started with a review of transactions, a reliability technology that had enormous impact, yielding multiple Turing Award winners, a thriving multi-billion dollar industry, and a wide range of remarkably robust solutions – within the constraints mentioned earlier. Attempts to shoe-horn problems that don't fit well within those constraints, on the other hand, met with market failures: products that have been rejected as too costly in terms of performance impact, awkward to implement or maintain, or perhaps too costly in the literal sense of requiring the purchase of a product (in this case, a transactional database product) that seems unnecessary or unnatural in the context where it will be used.

The discussion of reliable multicast pointed to a second, more complex situation. Here, a technology emerged, became quite real, and yielded some products that were ultimately used very successfully. Yet the area suffered a market failure nonetheless; while there are some companies still selling products in this space, nobody would claim a major success in the sense of transactional systems.

Our experience suggests that several factors contributed to the market failure for multicast platforms:

1. Multicast products have often been presented as low-level mechanisms similar to operating system features for accessing network devices. As researchers, this is natural: multicast is a networking technology. But modern developers are shielded from the network by high level tools such as remote procedure call – they don't work directly with the O/S interfaces. Thus multicast presentation has been too primitive.
2. These products were often quirky, making them hard to use. For example, our own systems from the 1990's had scalability limitations in some dimensions that users found surprising – numbers of members of groups, or numbers of groups to which a single process could belong, and performance would collapse if these limits were exceeded. Had we addressed these limitations, those systems might have been more successful.
3. The field advanced, first pushing towards object-oriented platform standards such as CORBA, and then more recently towards service oriented architectures such as Web Services. Multicast solutions didn't really follow these events – with the notable exception of the CORBA fault-tolerance standard. But the CORBA standard was a peculiar and constraining technology. It was limited to an extremely narrow problem: lock step replication of identical, deterministic server processes. This determinism assumption is limiting; for example, it precludes the use of any sort of library that could be multithreaded (an issue even if the application using the library is single-threaded), and precludes applications that receive input from multiple sources, read clocks or other system counters, etc. In practice such limits proved to be unacceptable to most users.
4. The products developed for this market were forced to target a relatively small potential customer base. The issue here is that replication arises primarily on servers, and at least until recently, data centers have generally not included large numbers of servers. Thus purely from a perspective of the number of licenses that can potentially be sold, the market is comparatively small.
5. These products have not made a strong case that developers who use them will gain direct economic benefit – a lower total cost of application development and ownership – relative to developers who do not use them. We believe that such a case can be made (as explained below), but this was not a priority for the research community and this has left developers facing a "reliability tax" – an apparent cost that must be born to achieve reliability.
6. Product pricing was too high. Here we run into a complex issue, perhaps too complex for this brief analysis. In a nutshell, to support the necessary structure around any product (advanced development, Q/A, support) a company needs a certain size of staff, typically roughly proportional to the complexity of the code base. Multicast is not a simple technology and the code bases in question aren't small or easy to test, particularly in light of the practical limitations mentioned in points 1 and 2, which made these products unstable for some uses. In effect, they are expensive products to develop and maintain.

Now consider prices from the customer's perspective. A data replication framework is a useful thing, but not

remotely as powerful a technology as, say, an operating system or a database system. Thus one should think in terms of pricing limited to some small percentage of the licensing cost of a database or operating system for the same nodes. But pricing for operating systems and databases reflects their much larger markets: a revenue stream is, after all, the product of the per-unit price times the numbers of units that will be sold. Thus multicast products are limited to a small fraction of the number of machines, and a small fraction of the product pricing, of a database or similar product – and this is not a level of income that could support a thriving, vibrant company. But if vendors demand higher per-unit licensing, customers simply refuse to buy the product, seeing the cost of reliability as being unrealistically steep!

7. These products have often demanded substantial additional hand-holding and development services. In some sense this isn't a bad thing: many companies make the majority of their income on such services, and indeed services revenue is what kept the handful of multicast product vendors afloat. And it may not be an inevitable thing; one can easily speculate that with more investment, better products could be developed. Yet the current situation is such that the size of the accessible market will be proportional to the number of employees that the company can find and train, and that most product sales will require a great deal of negotiation, far from the "cellophane-wrapped" model typical of the most successful software products.

In summary, then, the ingredients for a market failure are well established in this domain. The bottom line is that for solutions to ever gain much permanence, they would need to originate with the major platform vendors, and be viewed as a competitive strength for their products. This has not yet occurred, although we do believe that the growing popularity of massive clusters and data centers may shift the competitive picture in ways that would favor products from the vendors. The issue here is that building scalable applications able to exploit this price point involves replicating data so that queries can be load-balanced over multiple servers. This is creating demand for replication solutions – and hence opening the door for technologies that also promote fault-tolerance, security, or other QoS properties.

Yet, having worked with vendors of products in this area, we've also noted commercial *disincentives* for commoditized reliability. The issue is that many platform vendors differentiate their systems-building products by pointing to the robustness of their components. They offer the value proposition, in effect, that by executing an application on their proprietary product line, the user will achieve robustness not otherwise feasible. If end-users can build robust distributed systems without needing expensive reliability platforms, platform vendors might lose more revenue through decreased sales of their high-margin robust components than they can gain by licensing the software supporting application-level solutions.

And what of the theoretical work? Here, we believe that researchers need to recognize that theory has two kinds of markets. One is associated with the community of theoreticians: work undertaken in the hope of shedding light on deep questions of fundamental importance and of influencing future theoretical thinking. Impact on the commercial sector should not be used as a metric in evaluating such results.

But we also believe that the experience with practical Byzantine solutions points to an avenue by which the theory community can have substantial impact. The trick is to tackle a hard practical problem that enables a completely new kind of product. We believe this lesson can be applied in other settings. For example, real systems are stochastic in many senses. The research community would find a rich source of hard problems by looking more explicitly at probabilistic problem statements framed in settings where networks and systems admit stochastic descriptions. Results translate fairly directly to real-world settings and would be likely to find commercial uptake.

V. RECOMMENDATIONS

Not every problem is solvable, and it is not at all clear to us that the market failure in our domain will soon be eliminated. However, we see reason for hope in the trends towards data centers mentioned earlier, which are increasing the real value of tools, but want to offer some observations:

- Researchers need to learn to listen to consumers. On the other hand, one must listen with discernment, because not every needy developer represents a big opportunity. This is particularly difficult during dramatic paradigm shifts, such as the current move to net-centric computing. When such events occur, it becomes critical to focus on early visionaries and leaders, without being distracted by the larger number of users who are simply having trouble with the technology.
- We haven't made an adequate effort to speak the same language as our potential users, or even as one-another. For example, if our users think of systems in stochastic terms, we should learn to formalize that model and to offer stochastic solutions.
- Practical researchers have often put forward solutions that omit big parts of the story, by demonstrating a technique in an isolated and not very realistic experimental context. Vendors and developers then find that even where the technology is an exciting match to a real need, bridging the resulting gaps isn't easy to do. In effect we too often toss solutions over the fence without noticing quite how high the fence happens to be and leaving hard practical questions completely unaddressed. Only some vendors and developers are capable of solving the resulting problems.
- We need greater attention to our value propositions, which are too often weak, or poorly articulated. Technology success is far more often determined by

economic considerations than by the innate value of reliability or other properties.

- Our work is too often ignorant of real-world constraints and of properties of real-world platforms. For example, if a solution is expensive to deploy or costly to manage, potential users may reject it despite strong technical benefits.
- Our community has been far too fond of problems that are either artificial, or that reflect deeply unrealistic assumptions. The large body of work on compartmentalized security models is an example of this phenomenon.
- Real users seek a technology "process" not an "artifact", hence those of who develop technical artifacts need to be realistic about the low likelihood that conservative, serious users will adopt them.
- Concerns about intellectual property rights have begun to cloud the dialog between academic and commercial researchers. The worry is typically that an academic paper, seemingly unfettered by IP restrictions, might actually be the basis of an undisclosed patent application. If that patent later issues, any company that openly adopted the idea faces costly licensing. Hence companies either avoid dialog with academic researchers or limit themselves to listening without comment, lest they increase their exposure.

These observations lead to a few recommendations, which we focus on work aimed at the real world:

- Developers need to build demonstrations using real platforms if at all possible, and ideally to evaluate them in realistic scenarios.
- Results should make an effort to stress value in terms buyers will understand from an economic perspective. Even research papers should strive to show a credible value proposition.
- If the development team isn't in a position to provide long term product continuity, development and support, it should try to disseminate solutions via vendors, or to work with vendors on transitioning.
- Academic research groups should work with their University licensing officers to try to clarify and standardize the handling of software patents that the University might seek, in the hope that industry teams considering dialog with academic research groups will see IP ownership issues as less of a threat. The huge success of the Berkeley Unix project and its BSD licensing approach is a model that other academic research teams might wish to study and try to emulate.

VI. CONCLUSIONS

We've reviewed market forces that can have a dramatic impact on the ultimate fate of technologies for reliable computing, with emphasis on technical areas in which the authors have had direct involvement. Our review led to several kinds of insights. One somewhat obvious insight is that not all forms of academic research are of a nature to impact the commercial market: some work, for example, demonstrates the feasibility of solving a problem, and yet can't possibly be offered as a free-standing product because one couldn't conceivably generate a revenue stream adequate to support a sensible development and support process. Customers buy into a company's vision and process – it is rare to purchase a product and never interact with the company again. Teams that don't plan to create such a process shouldn't expect to have commercial impact. Yet academic groups are poorly equipped to offer support.

A second broad class of insights relate to the way that we pose problems in the reliability arena, demonstrate solutions, and evaluate them. We've argued that even purely academic researchers should pose problems in ways that relate directly to realistic requirements, demonstrate solutions in the context of widely used platforms, and evaluate solutions in terms that establish a credible value proposition.

A third category of suggestions boil down to the recommendation that researchers should ponder market considerations when trying to identify important areas for future study. This paper pointed to two examples of this sort –recent work on practical Byzantine agreement, and the exploration of stochastic system models and stochastic reliability objectives, arguing that these are both more realistic and also might offer the potential for significant progress. But many problems of a like nature can be identified. Others include time-critical services ("fast response" as opposed to "real-time"), scalability, and trustworthy computing (construed broadly to include more than just security).

Finally, we've suggested that the reliability community would do well to heal the divisions between its theoretical and practical sub-areas. While academic debate is fun, often passionate, and can lead to profound insights, we need to communicate with external practitioners in a more coherent manner. The failure to do so has certainly contributed to the market failures that have heretofore marked our field.

VII. REFERENCES

[1] K.P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications.* Springer; 1 edition (March 25, 2005)

[2] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems, 20(4):398--461, November 2002.

[3] MJ Fischer, NA Lynch, and MS Paterson. Impossibility of distributed consensus with one faulty process. JACM, 32(2):374--382, 1985.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1993.

[5] R Guerraoui, A Schiper. Consensus: The Big Misunderstanding. Proceedings of the 6th IEEE Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)

[6] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, 4 (3), pp.382-401

[7] D. Lomet. Process Structuring, Synchronization and Recovery Using Atomic Actions. ACM Conference on Language Design for Reliable Software, Raleigh NC. SIGPLAN Notices 12, 3 (March 1977) 128-137.