10 Things I Love and 10 Things I Hate about MCP Systems

2004 UNITE Conference
Philadelphia, Pennsylvania

Session MCP4050

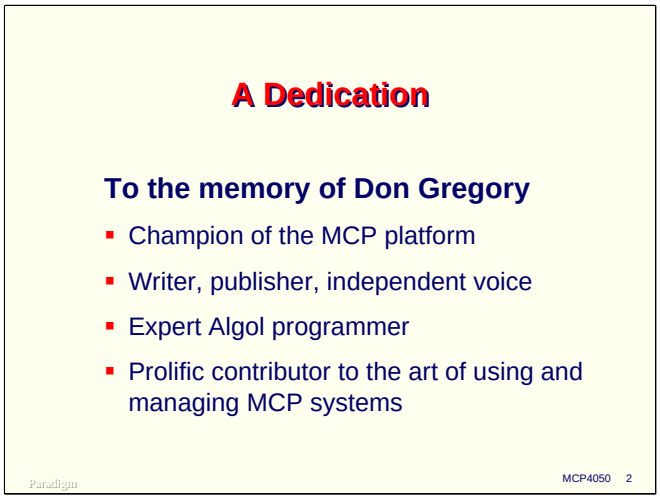
Monday, 20 September 2004, 2:45 p.m.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com



I will begin by dedicating this talk to the memory of Don Gregory. As many of you know, Don passed away unexpectedly this past year. Almost everyone who has worked in the MCP environment for any period of time knew Don, or at least knew of him.

Don was a tireless champion of the MCP platform, the MCP itself, and the Algol programming language. His interest in and dedication to this unique computing environment was such that he was motivated to write extensively about the platform and it capabilities. Principal among these were his classic series of Algol primers and his long-running *Technical Journal*. His practical advice on configuring and tuning MCP software and managing its ongoing operation remain second to none.

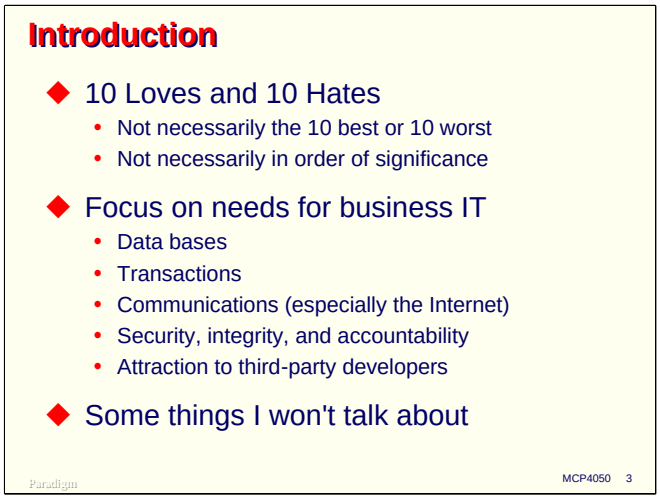
Don was also a prolific programmer and tool builder. Primarily through articles in his *Journal*, he developed an extensive library of useful programs and libraries, and bundled the license for their use with the publication's subscription.

Don was an ardent supporter of UNITE and the MCP user community, frequently presenting at the conferences and winning multiple Best Presentation awards.

Don was also a lot of fun, and I always looked forward to seeing him at UNITE and chatting with him. He would usually buy me a beer.

Don and I had a mutual regard, but we did not always agree. There are some things I am going to say in this presentation on which I'm quite certain Don would strenuously disagree. I know he would have listened, though, and tried to understand my arguments, and then made his counterpoints.

We will all miss Don. Directly or indirectly, his contributions to the MCP platform and community, and his independent voice, have benefited and will continue to benefit every user of this system. We will miss his energy, his expertise, his skill in communicating complex technical issues, and not the least his occasional candidacy for Governor of California.



I want to discuss today ten things about MCP systems that I really like and ten that I really do not like. Note that these are not necessarily the ten best or ten worst aspects of MCP, although many of my topics fall into those categories. The order in which they appear in this talk implies nothing about their rank among good and bad features. I have simply chosen ten each in an attempt to illustrate some really good characteristics of MCP systems, and some that are not so good.

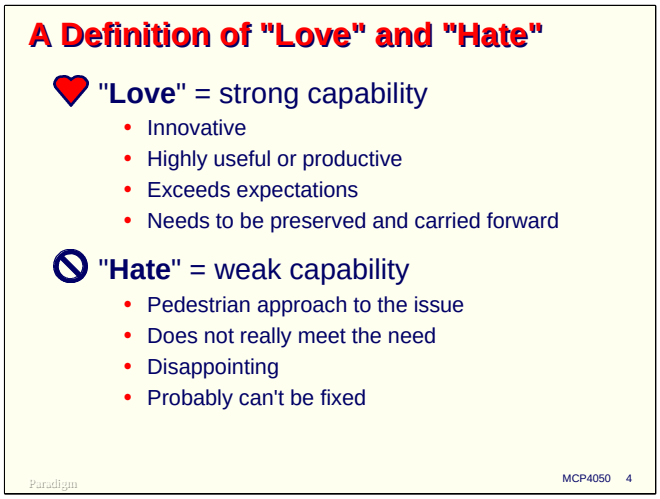
For the purpose of this presentation I am going to concentrate on typical needs of information technology for business. Computing has many facets, but most of us are here because we run our organizations, at least in part, on MCP systems. This use of systems for so-called "commercial" applications is one of the largest and most critical ones in computing.

To me, the primary characteristics of computer systems for business IT are:

- **Data bases**. We don't do that much actual computing in business IT. What we do mostly is tuck stuff away and pull it out again later. We use computer systems as repositories for our organization's data. Most organizations are now completely reliant on our computer systems as repositories.
- **Transactions**. Transactions are events, or more properly, a record of an event that takes place in our organization. There is typically some data that is associated with this event, and we need to store that data or use it to update other data in our repositories.
- **Communications**. Repositories and transactions don't have much use unless you can bring them together, so communications is another vital area of business IT. The rapid acceptance of Internet technology, and the favorable economics associated with gathering and disseminating data over it, have now made computer-based communications a critical part of essentially every business.
- **Security, integrity, and accountability**. As we are all painfully aware, the increasing role of computer-based communications in business, along with the rise of Internet, has given us new challenges to protect the organization's information assets, ensure their integrity, and account for access to those assets. Our current difficulty in doing so is one of the great failures of modern technology. The way that most computer systems are designed has a lot to do with this.
- **Attraction to third-party developers**. This final item is something that is not talked about enough. The days of every organization constructing and maintaining its own suite of unique, custom software for business operations is over, and has been for some time. Most organizations already use at least some packaged software. Therefore, in order to be successful, a computer system must be attractive to third party developers.

Finally, there are a couple of things I am intentionally not going to discuss. The first is anything outside the technical arena. There's lots you can say about Unisys – or any other vendor – in the areas of product strategy, marketing, support, business practices, and so forth. That is a discussion for another venue.

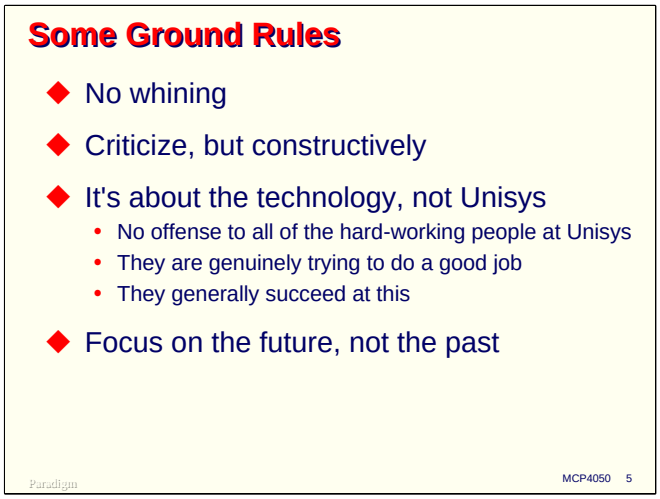
Second, there are some technical areas of MCP systems that I just don't know enough about to discuss in any detail, so whether or not they belong on my lists of ten, I've had to put them aside. Chief among these are BNA, LINC, and Distributed Transaction Processing (OLTP/DTP).



"Love" and "Hate" are good attention getters in a title, but they are strong words. I want to clarify here what I mean by them.

By "**Love**" I mean that something is a really strong capability. It may be very innovative, highly useful, exceeds expectations compared to other systems on the market, or some combination of these. The things I love about MCP systems are so good that I feel very strongly that their characteristics and benefits need to be preserved and carried forward into the future.

By "**Hate**" I mean that something is a capability with significant weaknesses. It may be something that is based on a pedestrian concept or approach, or does not really meet our needs, or is otherwise disappointing. In most cases, I don't see these things as having a weakness in implementation, but rather in concept or design. The things I "hate" about MCP systems probably can't be fixed, and in these areas I believe an entirely new approach is necessary.

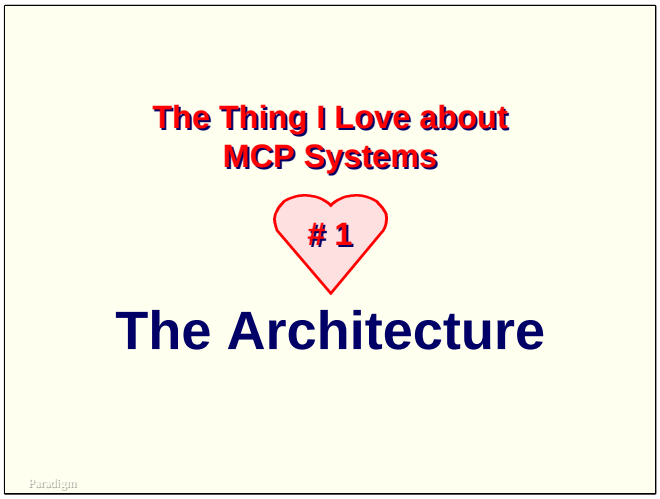


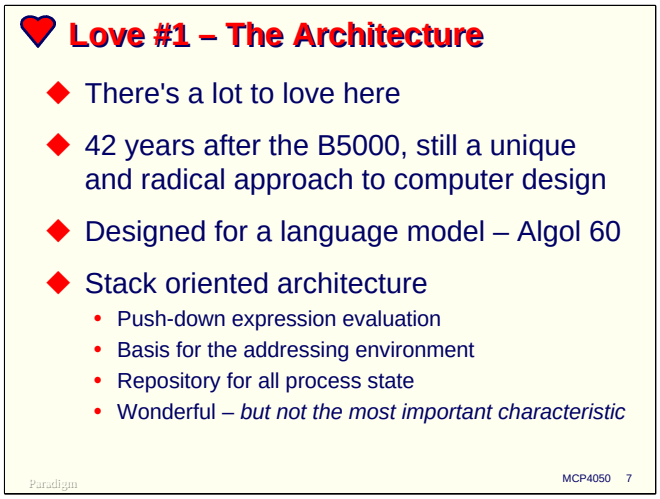
The majority of the information flow at UNITE has typically been from Unisys to the user community. I think this flow needs to be more balanced. We need to use the forum of UNITE to feed more information back to Unisys. We feed lots of information back to Unisys on a regular basis in the form of trouble reports and new feature suggestions, but those channels tend to have a very narrow focus and address relatively small issues.

The user community needs to be thinking and talking – both to Unisys and among itself – about the larger issues of its IT infrastructure and how Unisys products both help and hinder that infrastructure. I hope that others will begin to step up and make their contributions at this level.

Because this is something of an unusual topic for a UNITE presentation, I decided that I needed some ground rules as a guide to what is productive to talk about and what is not.

- First, no whining is allowed. The discussion should be substantive and open. Minor complaints are the province of trouble reports.
- Criticism should be encouraged, but it should be constructive criticism. The discussion should focus not on how bad something is, but how it doesn't meet needs and how it should be better.
- The discussion should be about the products and technology of Unisys, not Unisys or its people. Just because I categorize something as one of my "hates" does not mean I dislike the people responsible for it or necessarily think they are doing a bad job. Unisys, through its Burroughs lineage, is the only company to succeed in putting a really different computer system on the market and keeping it there for over 40 years. For that they deserve our admiration and respect. They certainly have mine.
- Finally, I want this discussion, and others like it that may come later, to focus on the future, not the past. It's necessary, of course, to understand the past and how we got to where we are today, but as a means to build a better future, not as an end. We need to talk about how things could and should be.


The Thing I Love about
MCP Systems
1
The Architecture
Paradigm



Although I said in the introduction that these topics were in no particular order of significance, the architecture of MCP systems easily ranks as my number one favorite.

There's a lot to love here. 42 years after the Burroughs B5000 was introduced, its direct descendents – the ClearPath systems we use today – still represent a unique and radical departure from mainstream designs. No other major architecture has been as ignored (and occasionally disparaged) by academia and the trade press as this one, and yet it lives on. It does so partly because of the major investment most of us have put into our applications. For many it has provided the easiest and least-risky migration path from other Burroughs architectures which are now obsolete. It also lives on, though, because of its strong points – reliability, resiliency, rich functionality, and ease of administration, among others. I suggest that all of these strong points are a direct consequence, in one way or another, of its underlying system architecture.

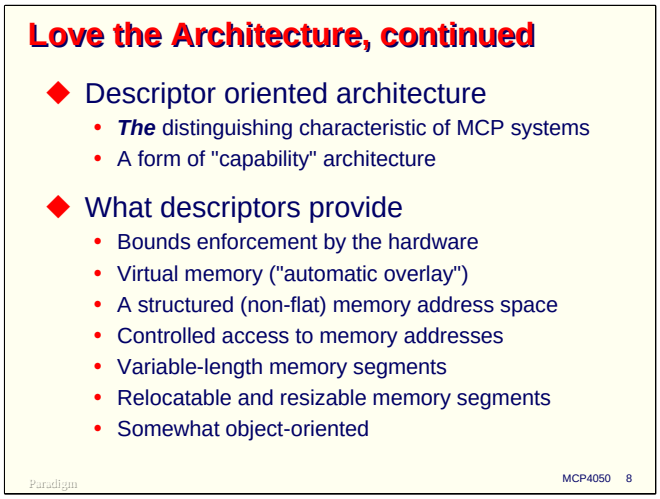
There are a number of aspects to architecture, but in this presentation I am mostly concerned with processor and memory design, their relationship to each other, and their joint relationship to applications. Certainly I/O is another important aspect, and one in which the large MCP systems excel, but I/O is secondary to the points I want to make here. Another distinction is between architecture and packaging. I consider the evolution from A Series through the NX-series ClearPath MCP systems to the current Libra series to be largely one of packaging. Aside from extensive changes in the I/O subsystem, there has been relatively little architectural change.

One highly significant – and highly unusual – aspect of the MCP system architecture is that it was originally designed for a specific language model, Algol 60. It is reputed to have been designed by a team of hardware and software people working together, but it's clear the software types had the major say. This Algol orientation is rife through the architecture, some of it giving us outstanding capabilities, and some of it inhibiting what the machines can do. Overall I am convinced this was the right way to go about designing a system, whatever disagreements people may have with this particular outcome.

The B5000 and its successors have usually been touted as "stack-oriented" machines, and this is true. Stacks are intrinsic to the architecture. On the systems with proprietary processors, stacks are manipulated and maintained directly by the hardware.

Stacks are used for the traditional purpose of push-down expression evaluation and the support of zero-address instructions, but they are also the basis for the addressing environment and are the repository for all process state. Every running task is associated with a stack, and the association between stacks and processes is so close that we use the terms interchangeably.

The stack mechanism is a wonderful capability, but to me it is not the most important characteristic. In fact, stacks are a distant second place to what I consider *the* outstanding feature of MCP systems.

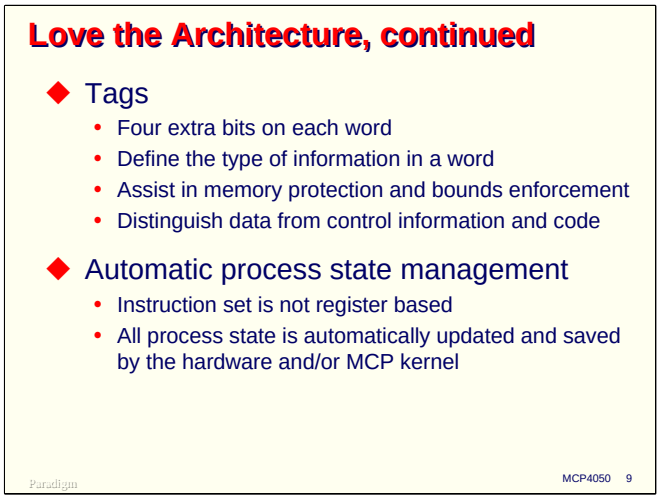


By far the most distinguishing characteristic of MCP systems – and to me, the most important – is their use of *descriptors*.

Descriptors are a special type of control word. Basically, they point to an area memory. Computers that manage access to memory through special control words are often characterized as having a "capability" architecture. With the possible exception of the IBM AS/400 (and its System/38 predecessor), MCP systems are the only commercially successful example of a capability design.

Like stacks, descriptors are intrinsic to the architecture and are supported directly by the hardware. Descriptors do so many things, it's a little difficult to enumerate all of them and describe how their multiple functionalities interact. Here's my list of the major contributions descriptors make to MCP systems:

- **Bounds enforcement**. Descriptors not only point to an area of memory, they describe its size. Anyone who has suffered an "invalid index" fault has descriptors to thank for this. Since it's embedded in the hardware, this bounds checking cannot be turned off, which makes it a major contributor to system security.
- **Virtual memory**. Descriptors are the direct means by which the hardware can detect whether an area of memory is physically resident or not, and if not, where it can be found on disk. While today's large memory sizes make overlay less important than it was 30-40 years ago, descriptors still play a critical role in allowing MCP systems to defer memory allocation until an area is actually referenced and to allow memory for individual tasks to reach an effective working set.
- **A structured memory address space**. Physical memory is a flat, monolithic space. Most other architectures implement virtual memory to emulate this flat address space. MCP systems have a much more structured approach to addressing memory. The way that memory is used depends on the programming language, but generally individual descriptors are used for individual language "objects." Only the operating system (and only small portions of it at that) work in the flat, physical space with absolute memory addresses.
- **Controlled access to memory addresses**. If there is one thing we have learned in computing – and in most cases keep not doing anything about – it's that giving programmers memory addresses is a lot like giving teenagers whiskey and car keys. Sooner or later they will abuse them, and usually it's sooner. Addressing of memory in MCP systems is very indirect, and is closely controlled by the hardware and operating system.
- **Variable-length memory segments**. Most other architectures chop their virtual memory address space into fixed-length chunks, all of the same size. Descriptors allow MCP systems to allocate both real and virtual memory space in variable-length segments, each exactly the size that is needed. This also allows the system to more easily achieve a working set of the objects that need to be present in memory.
- **Relocatable and resizable memory segments**. Descriptors allow the system to move memory areas in the physical space at will. This aids efficient use of physical memory. Every Algol programmer who has used the **`RESIZE`** statement knows how useful that capability can be. Descriptors are what make it both possible and relatively efficient.
- **Somewhat object oriented**. Finally, descriptors are a rudimentary form of object, and give MCP systems some characteristics of an object-oriented architecture. I'll talk quite a bit more about objects and architectures, and I will discuss why the one foot that MCP systems have in this direction is not nearly



Another relatively unique feature of MCP systems is their use of tagged memory. Tags are extra bits attached to each 48-bit word in memory. The current iterations of the architecture (Delta and Epsilon) use four bits for tags.

Tags define the type of information stored in a word. They assist in memory protection and bounds enforcement, and most importantly, distinguish data words from control words and from words containing object code.

Between the bounds enforcement afforded by descriptors, and the identification of word types by tag bits, MCP systems are much more difficult to break into and implant viruses and worms. It can probably be done, but the traditional approaches (e.g., buffer overflows) essentially just don't work on MCP systems.

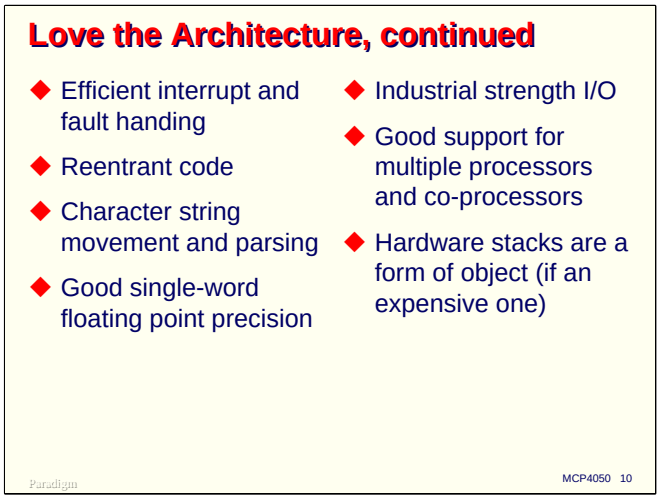
One of my favorite stories about the benefits of the MCP architecture comes from the days of the Nimda web server virus, in 2001. The attack vector of this virus is an extremely long HTTP query string. Many web servers at the time did not properly handle such long query strings in HTTP messages, resulting in overflows of the memory areas where they processed these strings. Two such web servers were the MCP's Atlas and Microsoft IIS. The difference in how these two pieces of software reacted to Nimda is telling.

- In IIS, running under Windows NT or 2000 on Intel processors, the long query string overflowed the area allocated for processing it and deposited data elsewhere in the IIS address space. This allowed Nimda to inject object code into the Windows environment and install the virus payload.
- In Atlas, which was no better prepared to deal with this phenomenon than IIS, the long query string also overflowed the area allocated to it. The bounds checking built into the hardware trapped this overflow, however, and the MCP DS-ed Atlas with a fault. The fact that the payload was Intel code rather than E-mode code is moot – the attack never got far enough for that to matter, and the system had still more protections (e.g., word tags for object code) to overcome before any virus payload could be made viable.

Something else which is under-appreciated in the MCP architecture (and which is sometimes incorrectly seen as a disadvantage) is its ability to automatically save its internal process state. The "state" of a process is generally composed of data in memory and processor registers. When the physical processor is switched from one process to another, or when the addressing environment changes (as in calling a procedure) at least part of this state must be saved and the destination environment's state restored.

The problem with this on most systems is that a lot of state management must be hard-coded into the application itself, which means that the programmer (or the compiler) has to do it. Since saving and restoring state can be expensive, there is a real temptation to take shortcuts. Also, sometimes you simply forget to do it, or to do it completely.

MCP processors have lots of registers, but essentially none of them are directly accessible to application code. Most of the registers have special purposes, and the hardware (along with small portions of the operating system) knows when they need to be saved and restored. Having all of this state management pushed down below the application level both enhances efficiency and helps prevent bugs.



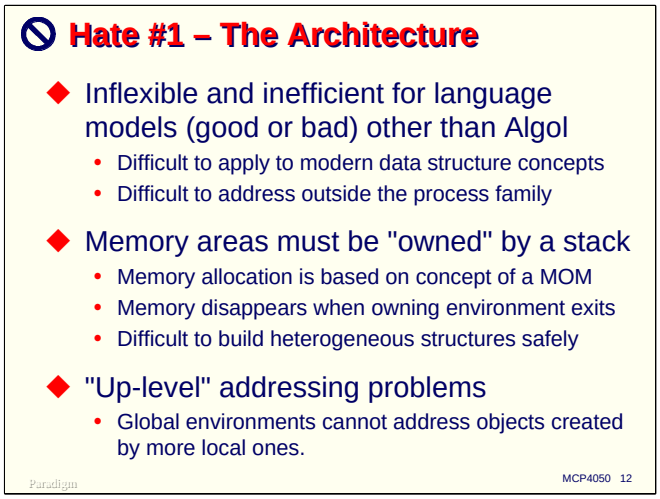
There are a number of other aspects of the architecture which deserve at least a brief mention:

- Interrupt handling and fault detection are nicely integrated with the stack mechanism. Interrupts are treated like a hardware-induced procedure call, which simplifies the design of the MCP and often results in a minimum of state saving and process switching.
- All object code is by definition read-only and is automatically reentrant. If you run two copies of the same object file at the same time, only one copy of the code is loaded to memory.
- As every Algol programmer knows, MCP systems have extremely powerful facilities for moving and parsing strings.
- The 48-bit word size gives the architecture good single-word floating point precision – over 11 digits. The 32-bit word size of many other systems gives them only about seven digits in single-precision floating point, requiring that calculations be done in double precision in order to get meaningful results.
- One thing that MCP systems do just about better than any other system is heavy-duty I/O. This is critical for both the transactional and batch workloads typical to business IT environments.
- The systems have extremely good support for multiple processors, and also handle co-processors (such as the I/O subsystem) very well.
- I mentioned earlier that descriptors are a rudimentary form of object. With libraries, you can use hardware stacks as a very capable form of object. The only problem is that this is a very expensive way to implement objects, and the architecture currently supports only a limited number of stacks.

The Thing I Hate about MCP Systems

The Architecture

Paradigm



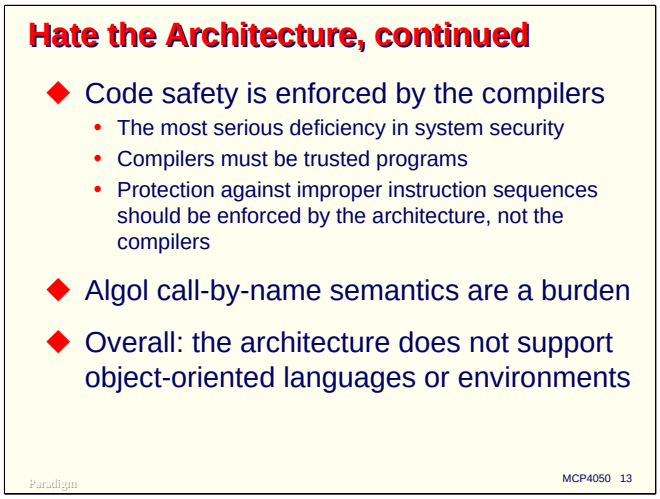
One major downside of an architecture that is so closely tuned to Algol is that it's not all that good for other languages. This problem is commonly discussed in terms of COBOL performance, especially in comparison to the V Series. Actually, MCP systems are not all that bad at COBOL. When you look at individual program performance, all but the latest large MCP systems look poor. When you look at performance of a *system* of COBOL programs, however, the benefits of the architecture begin to become apparent. The very things that made the V Series good for individual batch COBOL programs – a flat memory model and primitive addressing – eventually doomed it at the system level.

My main concern over the inflexibility of the architecture lies in another area, however. The main problem that I see is that MCP systems are limited to managing memory in the hierarchical, stack-oriented, Algol procedural model. It is difficult to extend this to the addressing and memory management requirements of more modern data structure concepts, where objects are allocated on demand and can be referred to from any number of places.

MCP systems have wonderful capabilities for sharing objects among the members of a process family, but it is much more difficult to share them outside of the family. The best we can usually do is make copies of things and send them as messages, e.g., using port files or V Series-like Storage Queues (STOQ).

A lot of this difficulty comes from the concept of the "mother" descriptor, or MOM. The ASD enhancements to the architecture in the mid-1980s mitigated many of the more serious problems with MOMs, but the problem still exists that memory areas must be owned by a stack (or a dope vector rooted in a stack), and the memory is deallocated when its owning environment exits. It's therefore impossible for an application to build safe heterogeneous structures (mixtures of data and descriptors) and to generate safe pointers to objects which are not within its current scope.

MCP systems deal with this limitation by prohibiting "up-level" addressing, i.e., a more global environment cannot maintain a reference to an object in a more local environment. The MCP does a great job of this, and in the context of Algol, it makes sense. In the context of more modern data concepts, though, it's a serious limitation.



Another problem with the MCP system architecture concerns object code. While tags and descriptors provide a great deal of system integrity and security, it is possible for certain instruction streams to violate security. Therefore, compilers have to be trusted programs. This is why we have the "MP" command to designate a program as a compiler.

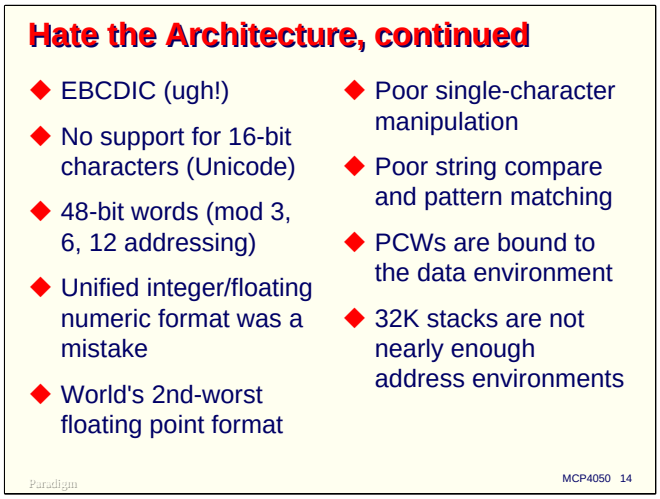
Enforcing code safety by restricting the programs that can generate code is a poor approach to system security. Ideally, it should not be possible to violate system security by improper instruction streams. The architecture itself should protect against this.

Yet another problem with the Algol orientation of the architecture is its support for "call by name" parameters. The Algol 60 report required that call-by-name formal parameters in a procedure be replaced by the corresponding actual parameters when the procedure was executed.

The idea of effectively replacing formal parameters by the text of their corresponding actual parameters was a radical one at the time, and has a number of impacts on the practicalities of implementation. Some thought that it would never be possible for Algol 60 to be implemented in a physical computer, partly due to the requirements of call-by-name semantics. The fact that the B5000 and it successors were able to implement this in hardware should be regarded as a triumph.

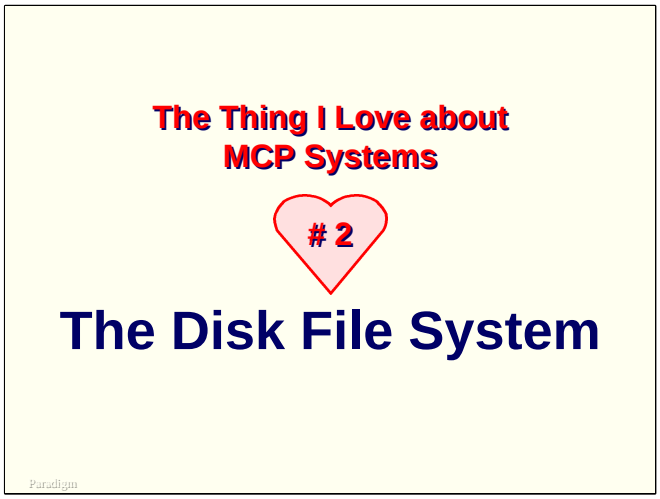
It turns out, though, that full call-by-name capability is actually seldom used, and in most cases there are now better ways to accomplish the same end. Call by name complicates the design of the processor and (among other things) inhibits memory look ahead. It could probably be dispensed with, at least at the hardware level.

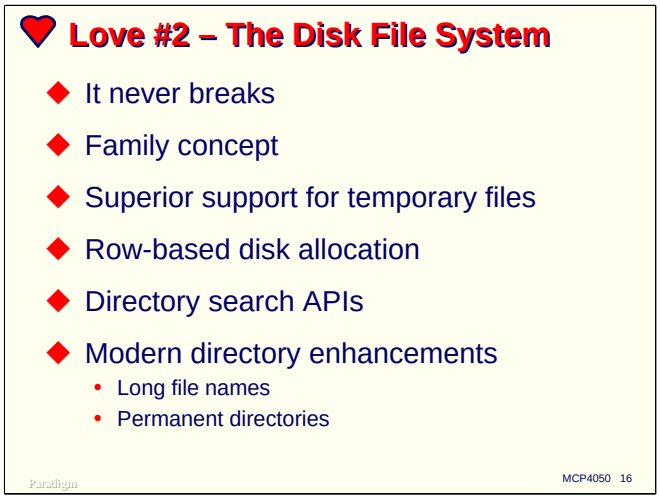
Overall, the main point I want to make here is that, while the MCP system architecture is an amazing conception with many beneficial aspects, much of it gets in the way of supporting modern software design and data structuring concepts. In particular it is not a good system for the support of object-oriented languages and environments.



I have a number of lesser issues with the architecture, which I will mention just briefly:

- EBCDIC is a terrible character coding scheme. It's perfectly understandable why in the mid-1960s Burroughs chose to go with EBCDIC rather than ASCII, but I wish they hadn't.
- The architecture has no built-in support for 16-bit characters. Support for Unicode is becoming increasingly important.
- The 48-bit word size is another mid-60s decision that is understandable in retrospect, but now gets in the way. It worked well with six-bit characters, but the advent of eight-bit characters has saddled with architecture with the need to divide by factors of three when computing addresses. This is hard to do efficiently, compared to dividing by powers of two.
- The B5000 adopted a unified integer and floating point numeric format. That same format has been retained through to the ClearPath line. It must have seemed like a good idea in the late 1950s and early '60s, but having the hardware decide at run time what's an integer and what's a floating point number is not very efficient. It really drags down integer arithmetic with the need to integerize results every time before storing them.
- Speaking of floating point, I consider the format used by MCP systems to be among the world's worst, second only to that used by the IBM 360. A large part of the problem is the base-8 exponent, which causes 3-bit shifts in the mantissa and contributes to loss of precision during rounding. The IBM format is worse because it uses base-16 exponents. IEEE 754 floating point has a much better design in this respect.
- The architecture has excellent character stream manipulation capabilities, but in contrast, its ability to do character-at-a-time manipulation seems poor. The Level Delta architecture introduced character load and store instructions, but support for these does not appear to have migrated into the languages, particularly Algol.
- Program Control Words (PCWs) are typically used as addresses into object code segments. Unfortunately, they must be bound individually at run time to a data environment, which further inhibits the architecture's applicability to object-oriented environments.
- Current ClearPath models can support tens of thousands of stacks. This is plenty for the traditional uses stacks are put to, but if you start trying to use stacks as objects, 32K (or 64K, or even 256K) stacks are not nearly enough. Stacks are too expensive to use as general objects, anyway, so an entirely different mechanism is needed.


The Thing I Love about
MCP Systems
2
The Disk File System
Paradigm



The second thing I love about MCP systems is their disk file system.

The thing I love most about the file system is that it never breaks. I have seen hardware problems cause it to lose files, and of course disk crashes cause you to lose the whole disk, but in the 28 or so years since it was redesigned for the Mark 2.7 release, I have yet to see the disk file system itself fail beyond the point of recovery.

Another thing I like is the family concept for disks. This nicely isolates the physical disk subsystem from application software, and does so in a convenient and extensible manner. Compared to "drive letters," it's wonderful. Over the years it has easily accommodated RAID and SAN-based disk subsystems. My one wish for the implementation of families is that they could have a granularity lower than that of physical disk units, but with current SAN technology, that is not much of an issue any more.

I think the way that the MCP isolates disk space management from the directory is also a good idea, and gives us support for temporary disk files that is superior to any other system that I've seen.

The row-based disk allocation mechanism used by the MCP is a good compromise between the needs of space efficiency and address translation efficiency, but giving the advantage to address translation. Checkerboarding of available space is a problem with this approach, although it is less so with today's large disks.

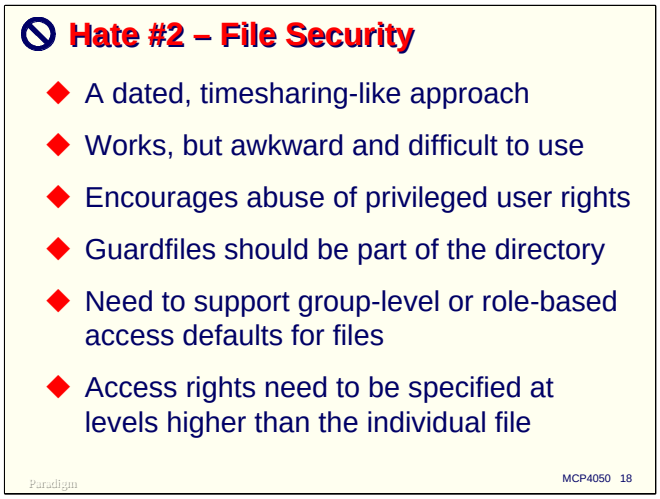
Directory search is also a strong point of the disk file system, especially since the ASERIES_INFO and PDIR-like interfaces were implemented over ten years ago. I really wish, though, that files were maintained in the disk directory in name order instead of name-size order, and that the directory search APIs were easier to use and directly available to languages other than Algol.

Finally, the modern enhancements to the directory mechanism, particularly long file names, are most welcome. It would be better, however, if Unisys could finish the job and fully integrate long file names and permanent directories with the legacy tools and utilities.

The Thing I Hate about MCP Systems

File Security

Paradigm



One aspect of the disk file system that I actively dislike is file security. It's not that it doesn't work – it does – it is just not based on a very good model. What we have today is an outgrowth of a dated approach that was originally designed for timesharing environments. This isn't the way that MCP systems are used anymore.

There are a number of areas where I find the file security mechanism awkward and difficult to use:

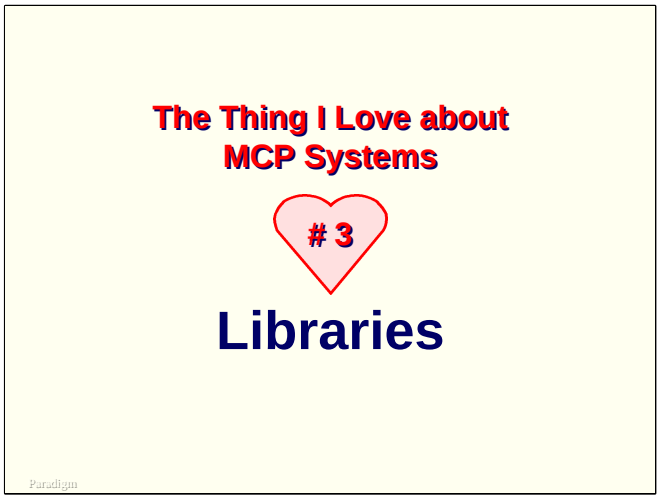
- Security must be applied to files individually. There is no way to define in advance – based on, say, the directory structure – what security should be applied to a file. It must be applied specifically to each file, and only during the file's creation or after the file has been created.
- Requiring the usercode as a file prefix is another characteristic that comes from the timesharing security model, and in retrospect, I think it was a mistake. File ownership should be established separately from its naming convention.
- Generally, non-privileged users are restricted to creating files only under their usercode.

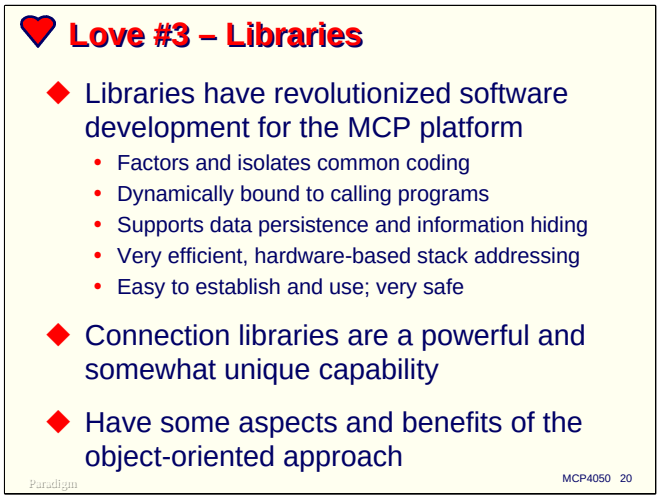
One serious consequence of the poor security model is that it encourages abuse of privileged usercodes. I have been to too many sites where everyone is a privileged user, simply because it's too difficult to work around the awkwardness of the file security mechanism. This is not a good thing for a system targeted at enterprise-level IT.

Guardfiles are an excellent capability, but they have been essentially tacked on the side of the disk directory and file security mechanisms. Being able to associate detailed lists of fine-grained access rights to a file is necessary, but it should be an integral part of the directory mechanism, not stored in a separate file.

Default access rights need to be associated with individual users and groups of users, or perhaps based on user roles. This should be supported directly in the file security mechanism. POSIX has some capabilities in this area. Novell Netware and Windows NTFS are good places to look for better models, especially with respect to roles.

Access rights also need to be specified at levels higher than that of individual files. Most operating systems allow you to do this at the disk directory or folder level, which is a good model to start from, but there are probably even better models than that based on user roles.


The Thing I Love about
MCP Systems
3
Libraries
Paradigm



Another of my loves that is very high on the list is libraries. Libraries are programs that export subroutine entry points which other running programs can call. Libraries are one of the few ways that MCP tasks can access procedures and data outside of their process family.

Libraries have revolutionized software development and maintenance for the MCP platform, both in system software and for user-written applications. Libraries have the following major advantages:

- The principle use of libraries is to factor common coding and to isolate it in separate codefiles. This helps both in modularizing a design so that developer resources can be focused on specific elements and in on-going maintenance of the resulting software.
- Libraries are dynamically bound to calling programs. The association between a caller and a library is made when (and only if) an entry point in the library is called. The primary advantage of this late binding is that changes to a library are available immediately and are activated the next time a program links to it. If necessary, a calling program can disconnect and reconnect to a library to obtain the latest version.
- Because libraries are a form of system process (task), they have their own data environment. This allows them to act as a data sharing mechanism among a number of other, otherwise unrelated processes. Libraries can manage the persistence of data, independent from the lifetimes of its calling programs. Libraries are also a powerful information-hiding mechanism. Libraries can be used as a wrapper around a resource (such as a data structure or a communications protocol) to provide a convenient, safe interface to that resource, hiding the grubby details.
- Libraries as implemented on MCP systems are very efficient and make extensive use of the hardware-based addressing mechanism for stacks. There is a small amount of MCP overhead to set up linkage between a caller and a library when the library is first referenced. After that, subroutine calls and the changes in addressing environment are handled entirely by the hardware.
- Libraries are very easy to create and very easy to use. Libraries can be referenced by most languages supported in the MCP environment, and can be created in all of the major application languages.

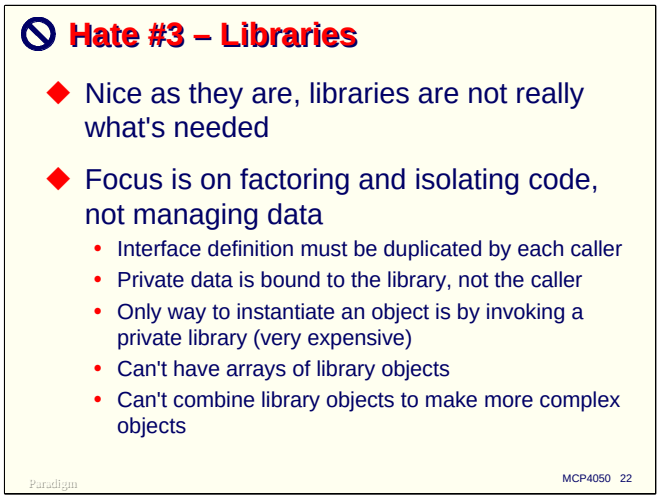
The original implementation of libraries are now called "server" libraries. About ten years ago, Unisys implemented an additional form of library for Algol and NEWP, called "connection" libraries. These implement a more powerful metaphor than server libraries, allowing one program to act as multiple libraries simultaneously, allowing libraries to call each other by establishing "connections" between them, and supporting per-connection instances of data which can be efficiently addressed within the library, often avoiding the overhead of table indexing. Connection libraries appear to be a unique capability – I know of no other system that has anything like it.

Libraries also have some of the aspects found in object-oriented designs, and properly used, many of the benefits of an object-oriented approach as well.

The Thing I Hate about MCP Systems

Libraries

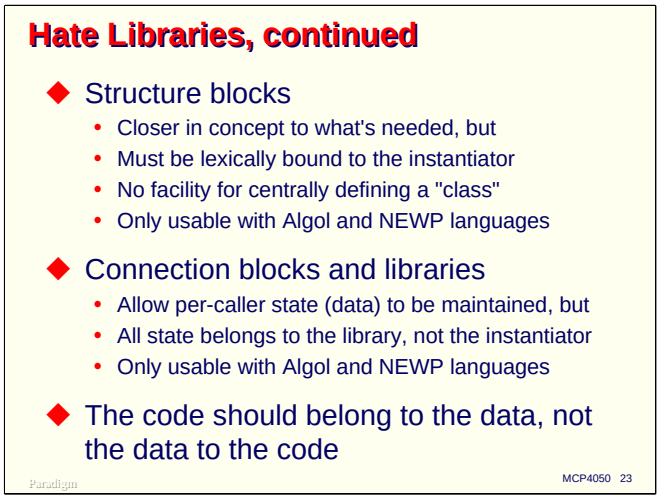
Paradigm



Libraries are a very powerful and efficient capability, but unfortunately they are not really what is needed.

The biggest problem with libraries is that their focus is on factoring and isolating code, not on managing data. While libraries have some aspects of object-oriented mechanisms, they fall seriously short in a number of areas:

- The definition of the interface to the library entry points must be duplicated manually by each caller. Libraries do not expose their interface, at least in any way that can be used by the compilers and application programs. The only way you can know if you've defined the interface correctly is to try to connect to the library's entry points – and suffer termination if you're wrong.
- All private data belongs to the library, not to the caller.
- The only way you can instantiate a copy of an object defined through a library is to invoke a private library. This is not so bad if you only need a few copies, but it's very expensive to scale beyond that. Each private library requires its own stack. Stacks are relatively hefty data structures, and only a limited number of them can be supported by the system at a time.
- It's not possible to have arrays of objects implemented through a library.
- What's worse, there isn't any general way that you can combine objects implemented through a library to make more complex (and more useful) objects.



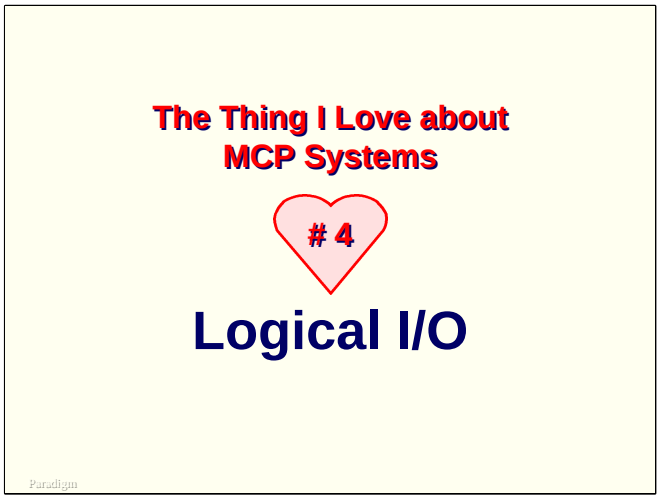
Structure blocks come much closer to what is needed, but structure blocks have three serious limitations:

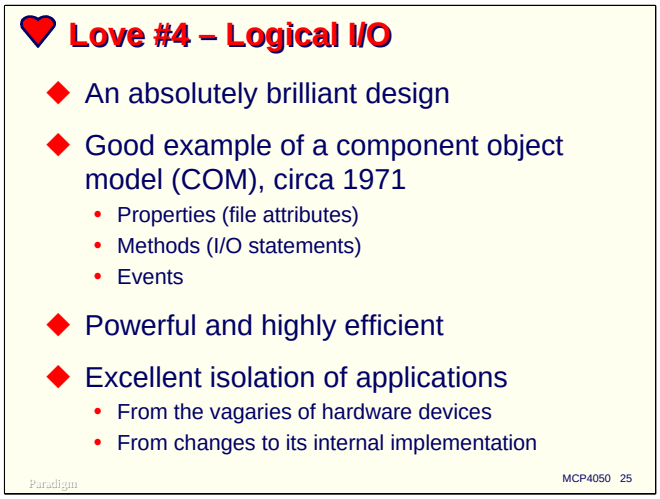
- They must be lexically bound to their instantiator. In other words, they have to be coded inside the program that invokes them.
- There is no way to define a structure block externally and use it as a class-like template for creating objects. You can, of course, store the structure block definition in an include file, but it's still lexically bound, and changes to the "class" definition won't be picked up by the programs until they are recompiled.
- Structure blocks are currently implemented only in Algol and NEWP.

Connection libraries combine the concepts of server libraries and structure blocks, which yield some really interesting and useful capabilities. When used in a connection library, structure block-like entities are called "connection blocks." Connection libraries, however, still fall short of what object-oriented systems can provide:

- Connection libraries make it much easier to maintain separate state (data) for each caller of the library, but the data belongs to the connection library, not to the instantiator.
- Like structure blocks, connection libraries can only be created in Algol and NEWP. What's worse, they can only be accessed by Algol and NEWP callers.

We have server libraries and connection libraries because that is what can be safely and efficiently implemented using the MCP system architecture. Unfortunately, they address the wrong end of the problem. Libraries are bodies of code that own the data that they manage. What we need is exactly the opposite – bodies of data that own the code that manages them.


The Thing I Love about
MCP Systems
4
Logical I/O
Paradigm



Logical I/O is one of the most incredible pieces of software design I have encountered. The concepts on which it is based are very strong, but the initial implementation for the B6500, in the late '60s, was poor. Burroughs wisely kept the concepts and completely reengineered the implementation in 1971, and the result is essentially what we have today.

One amazing thing about Logical I/O is that it is a very good example of a component object model, but done in 1971, well before that concept was widely appreciated. It has the characteristics you would expect of a full object model – properties (which we call file attributes), methods (the I/O statements in the application languages), and for port and remote files, even events to signal real-time activity.

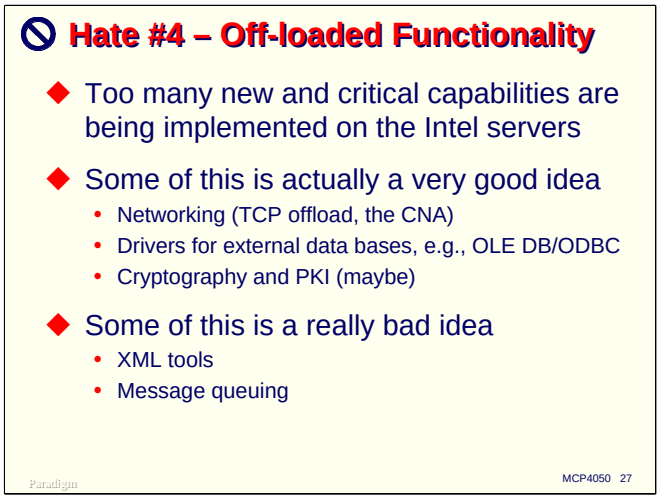
Logical I/O is very powerful and highly efficient. Like libraries, it capitalizes on the stack-based addressing mechanism to provide a simple API for a very complex internal process. Logical I/O has also been able to keep pace with changes in I/O devices and file techniques. Stream files are an example of something new which fits in quite nicely within the existing Logical I/O structure. The relatively new "virtual file" mechanism is something else that has been implemented without disturbing the structure or behavior of existing I/O capabilities.

Logical I/O does something else that I think is under-appreciated. It nicely isolates applications from the vagaries of hardware devices and from version-to-version changes to its internal implementation.

The Thing I Hate about MCP Systems

Off-loaded Functionality

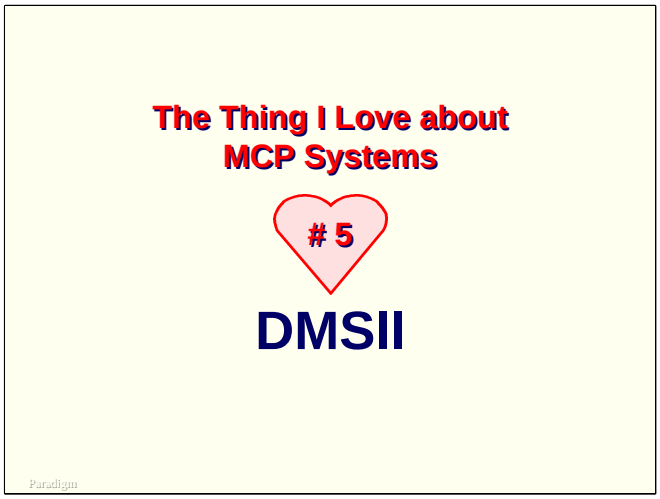
Paradigm

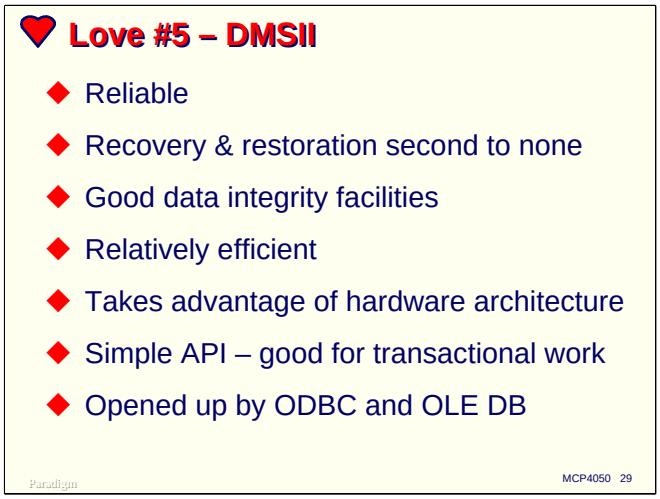


What do I mean by "off-loaded functionality?" That's simply all of the MCP stuff that runs on the Intel side of a ClearPath system.

Actually, off-loading some of this functionality is a very good idea. A prime example of good off-loading would be TCP/IP protocols. This effectively uses the Intel box as a front-end processor, a concept which is more fully realized in the ClearPath Network Appliance. I would also classify as good choices those things that are widely implemented in the Windows environment which would be too costly or impractical to reproduce for the MCP environment. Examples of this would include OLE DB and ODBC drivers for external data bases, and perhaps cryptography and public key infrastructure.

On the other hand, some of the off-loading we are seeing is, in my opinion, a really bad idea. Chief among these is the current XML implementation. XML is a critical component of future application design, and support for it needs to be built internally into MCP systems, where MCP-based applications can access it for their own purposes, and not just as an external interface to DMSII. Message queuing is another good example of functionality which should be implemented in the MCP environment, rather than exported to an external processor.


The Thing I Love about
MCP Systems
5
DMSII
Paradigm



DMSII is celebrating its 30th anniversary this year. It's a great product, and one of the principal strengths of MCP systems.

Its principle strength is reliability. DMSII supports what I think of as the primary requirement for a data base management system – when you give it data, it gives it back – on demand, every time.

The recovery and data restoration capabilities of DMSII are second to none. I think even Oracle and Microsoft SQL Server fall short of DMSII in this area.

DMSII also has quite good data integrity features, such as checksum and address check words in data blocks. You can choose to turn these on or leave them off on a data base-by-data base and structure-by-structure basis as your needs dictate.

DMSII is efficient, especially for transaction processing applications. It is not as efficient for ad hoc query and reporting, I think largely due to its one-record-at-a-time API design. This could be improved by having a much higher-level query interface, such as a query language that could return record sets.

DMSII is implemented with a precursor to library technology, and as such it takes excellent advantage of the stack-based addressing mechanism that is built into the architecture. The linkage between programs and DMSII is very efficient.

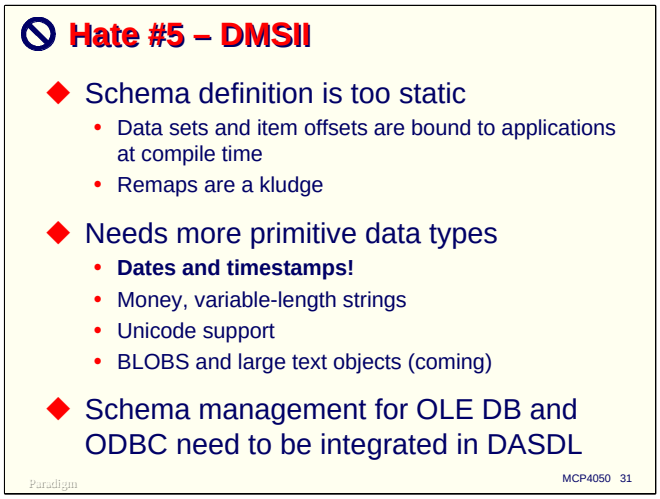
DMSII provides a very simple, straightforward API for application programs. This API is good for transactional work, where relatively few records need to be accessed per transaction, but is less efficient and more tedious to use for large or complex data retrievals.

One really nice addition to DMSII over the past several years has been the open access to it from external systems provided, first by ODBC, and more recently by the newer OLE DB mechanism.

The Thing I Hate about MCP Systems

DMSII

Paradigm



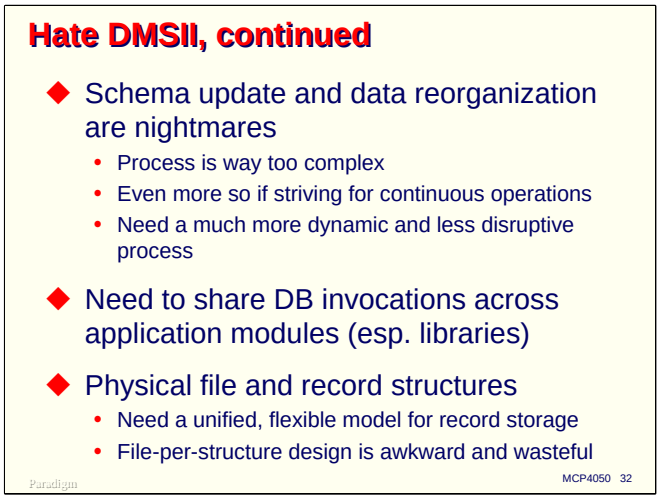
As useful and reliable as DMSII has proved to be, it is far from perfect. I think it has been lagging in a number of areas behind other modern data base systems.

One of my principal frustrations with DMSII is that its schema definition (DASDL) and the binding of that definition to applications is far too static. Data sets, indexes, and items within data sets are bound to applications at compile time. This may have been desirable – or even necessary – for adequate performance when DMSII was first designed, but it's no longer the case, as modern relational data base systems have shown. Remaps were developed in an attempt to mitigate this static binding, but they are something of a kludge and don't serve the purpose very well.

My main concern with static binding is that it gets in the way of an application's ability to respond dynamically to changes in requirements. With a lot of careful planning you can often make significant schema changes without seriously disrupting a running application, but it's so much work that a lot of sites don't bother unless they really have to. This is not a characteristic that will attract third-party developers to the platform.

DMSII desperately needs some additional data types. Chief among these are **date** and **timestamp** items. These types of items are vital to absolutely every business application, and it's inexcusable for a major data base management system not to support them directly. DMSII should also support things like money and variable-length strings. Support for Unicode is of increasing importance, especially in international markets. Support for very large data items, such as BLOBs and large text objects is needed, and Unisys is currently working on that.

The importance of open access to DMSII data from outside the MCP environment is steadily increasing. The Unisys OLE DB and ODBC facilities provide good capabilities in this area. They also support additional data types, especially dates and timestamps. Unfortunately, identifying these additional types to the open providers must be done manually, and in a process separate from the standard schema maintenance through DASDL. This is at least inconvenient, and potentially a source of serious problems. Jim Cogan proposed some years ago that the additional type information be made a part of DASDL so that the open providers would be able to pick up this information automatically. I think he is right about this, but it is the minimum solution. Actually implementing richer types in DASDL and making them available to applications through the API is a better long-term solution.



Unisys has done a fantastic job over the history of DMSII with the development DASDL update and reorganization. I remember, when on-line reorg first came out, being simply amazed that they could even do this with DMSII. The recent enhancements for ReorgDB are just the latest in this long list of enhancements to support schema update and data reorganization.

That said, the process of updating a DMSII schema and reorganizing data structures is a nightmare. I think anyone who has maintained a production DMSII data base for a while and then worked with a modern relational data base (e.g., Microsoft SQL Server) is floored by how easy it is to make changes to the schema and how well the DBMS can reorganize the data on the fly. On the other hand, anyone experienced with relational data bases who is coming to DMSII for the first time is probably appalled at what we have to go through for trivial schema updates.

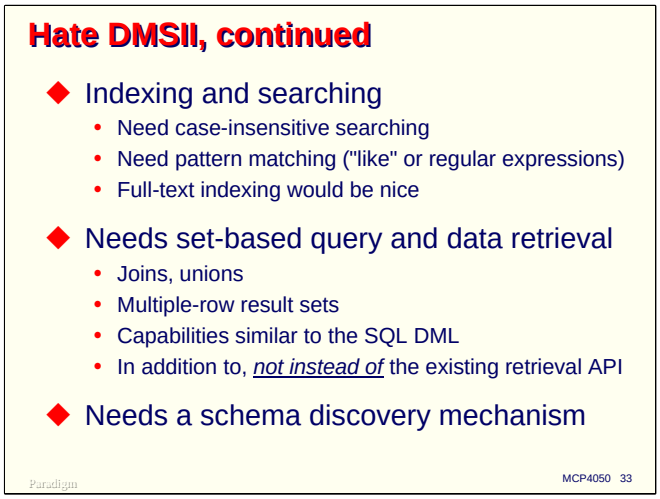
The approach DMSII uses for schema update and data reorganization is much too complex and much too disruptive to operations, especially if they have a high uptime requirement. That update and reorg are this way are direct consequences of DMSII's architecture, so this is not an easy thing to fix, if it's even possible.

As with static binding, my concern in this area is that it gets in the way of the system's ability to support the increasingly dynamic nature of business applications. We need a DBMS that is much more responsive to changing requirements, especially if the platform is to attract third-party developers.

One very limiting aspect of the DMSII API is that data base invocations are wired to a particular stack, and the presumption is that that stack (and perhaps its child processes) will do all of the data base operations against that invocation. This is all right for old-style batch processing application designs, but makes it extremely difficult to construct a transaction system as a series of functional modules and to separate the user interface logic from business rule logic. What we need is the ability effectively to pass a data base invocation as a parameter to procedures and libraries so the business rules can be maintained separately and factored into functional pieces. Separately implemented modules should be able to participate in the same data base transaction. This is another thing that probably cannot be addressed in the context of the current DMSII design.

Another area where I think the original design of DMSII has not held up well is its physical model for data storage. We have a variety of types of data sets, which are largely distinguished by how they format records in blocks. I think this is unnecessary. I would like to see a unified model for record storage, perhaps more along the lines of COMPACT data sets, with adjustments as necessary to support the needs of the ORDERED and RANDOM access methods. The goals here should be both more efficient data storage (and therefore more efficient I/O) and better support for dynamic schema changes.

I also dislike the use of a physical file for each data set and index. This is wasteful, especially for the modern practice of using many small tables for dictionaries of coded values. The DBA should be able to decide how data sets are grouped (or not) into physical files. Similarly, the grouping of index structures with their data or with each other should be something the DBA controls. The DBA still needs to control the physical attributes of files (blocksize, areasize, etc.) instead of the one-size-fits-all approach that other data base systems typically use.

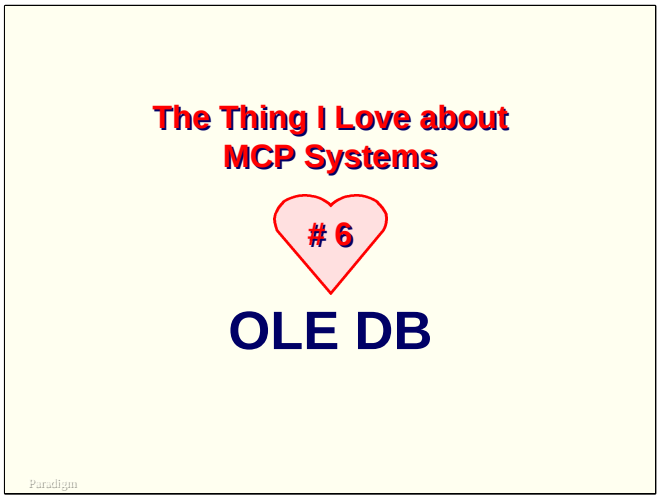


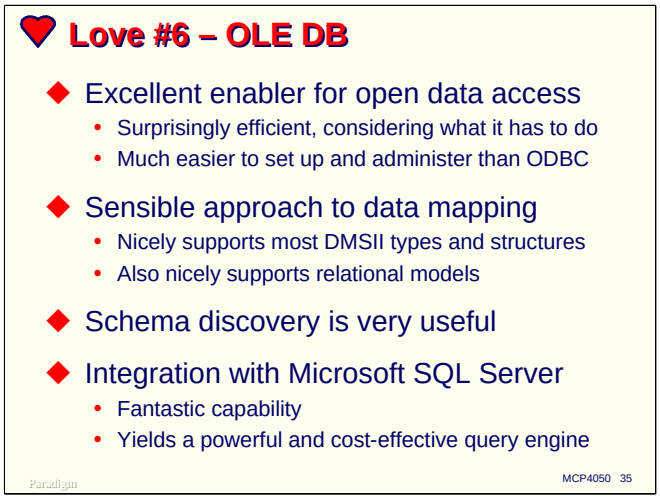
One of my major disappointments with DMSII is its limited and very literal-minded approach to indexing and searching. Chief among the problems here is the difficulty in maintaining an index to mixed-case text fields. The only practical way to do this at present is to define an additional data item that contains a normalized key value (e.g., translated to all upper case) and to use this additional data time as the index key. Maintaining a duplicate field for a normalized value is a huge burden on the application. At a minimum, the DBA should be able to define an index as case insensitive and have the data base software deal with normalization issues.

Beyond that limitation, DMSII selection expressions are very primitive. They can only use one index structure, and are suitable only for exact match or finding the beginning of a range of records to retrieve. More sophisticated expressions are needed, along with richer predicates, such as "like" matching or regular expressions. In addition, full-text indexing would be nice, especially for the new large text objects which are coming.

I would argue that DMSII needs much more than the current ISAM-like index search and retrieval mechanism it supports. This is good for transactional work, and should be retained, but for ad hoc query and reporting, set-based retrieval (such as you find in relational data base systems) is a much better way to go. Not only is this easier for the application to specify and use, but there is potential for much more efficient processing, since the work of joining tables, searching, and building a result set can remain inside the DBMS rather than spread across numerous low-level API calls between the application and the DBMS. Anyone who has experience with the SQL Data Manipulation Language (DML) understands how powerful this approach to query and retrieval can be. Something like the SQL DML (but not necessarily identical to it) is needed in DMSII. DMINTERPRETER simply does not fill the bill in this area.

Finally, DMSII could use a schema discovery mechanism. This is necessary to support third-party tools. OLE DB and ODBC implement this nicely, but it's only available outside the MCP environment. The only practical way to do this inside the MCP environment is to read the data base's Description file, which is a complex and version-dependent endeavor.


The Thing I Love about
MCP Systems
6
OLE DB
Paradigm



A relatively new thing about MCP systems that I really love is the OLE DB Provider for DMSII. This is an excellent enabler for open data base access. It is surprisingly efficient, considering what it has to do to meet the Microsoft OLE DB specifications and map OLE DB concepts to DMSII concepts. I have found OLE DB to be very easy to set up and administer, much more so than the ODBC provider for DMSII. The only significant inconvenience is having to adjust the schema for richer data types, principally dates. As discussed earlier, this is a problem with DMSII itself, not OLE DB.

Another thing I like about OLE DB is its very sensible approach to mapping DMSII data items and structures to its concepts of rowsets and fields. By backing off the relational model a bit, and creating the concept of hierarchical recordsets, OLE DB easily supports occurring items and embedded structures in DMSII. I also like the way it handles variable-format data sets – typically a bane in ODBC mapping strategies. Of course, OLE DB was primarily designed with relational structures in mind, and if the DMSII data base is already in a normalized form, the mapping works even better.

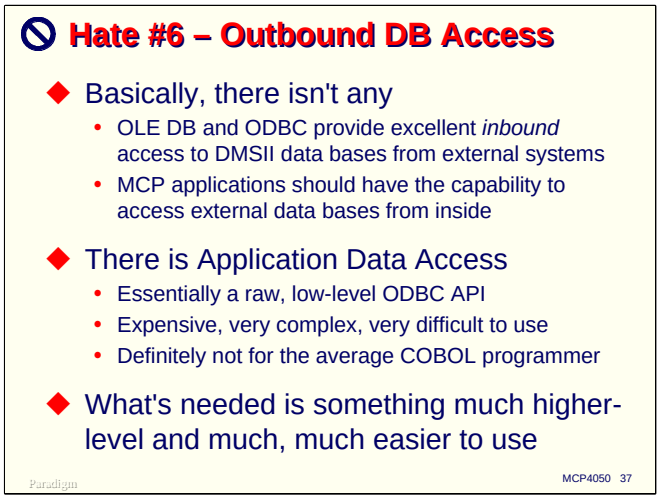
I have found the schema discovery capability in OLE DB to be very useful. Through the ADO wrapper for OLE DB and its schema discovery mechanism, it's quite easy to build tools in Visual Basic or VBScript that tailor themselves to individual data bases, DMSII or otherwise. Schema discovery is one of those things you don't know you need it until you have it.

Finally, one of the best parts about OLE DB is the way it enables Microsoft SQL Server to be used as a front-end query engine for DMSII. This is a fantastic capability. It's extremely powerful, relatively efficient, cost effective, and a workable substitute for the lack of a good query interface in DMSII, provided you only want to perform SQL queries from outside the MCP environment.

The Thing I Hate about MCP Systems

Outbound Data Base Access

Paradigm



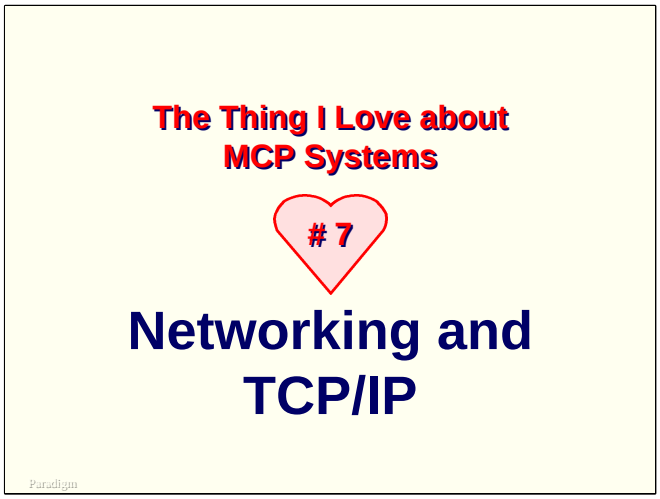
Related to OLE DB and open data base access, one thing I hate about MCP systems is the facility for outbound data base access. This is the ability to access external data bases from within the MCP environment. The thing I hate about it is that basically there isn't any.

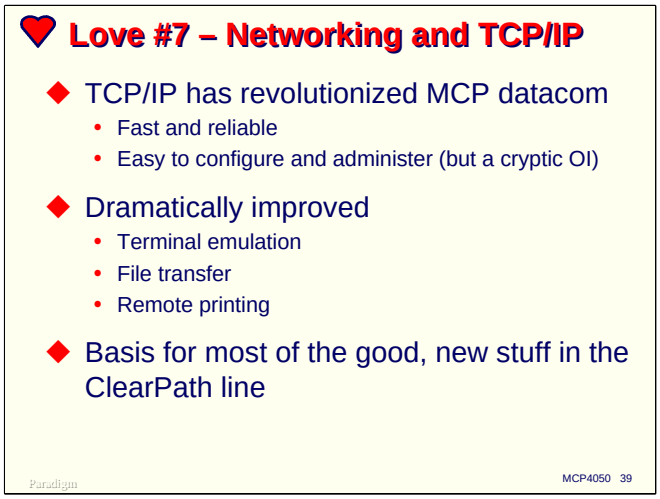
OLE DB and ODBC provide excellent access from external systems to DMSII resources. This facility should be symmetric – MCP-based applications should be able to access external data bases and participate in their transactions just as easily and efficiently.

There is an MCP product named Application Data Access that provides at least some of this capability. It is essentially a raw, low-level API for ODBC. The software in the MCP environment forwards API calls across a network to a Windows system, where a service translates them to calls on standard Windows ODBC drivers for the destination data base. The destination data base could be on that Windows system or elsewhere on the network.

Application Data Access must be licensed separately and is relatively expensive. The API is complex and quite difficult to use. It is definitely not something to expect an average COBOL programmer to master, at least not easily. For these reasons I cannot accept Application Data Access as a suitable solution for outbound data base access.

What is needed is an interface that is much higher level and much, much easier to use. Something along the lines of Microsoft ADO would be very nice, or perhaps an API that could map OLE DB or ODBC interfaces into something that looks more like the DMSII API. In any case, it needs to be something that can be readily invoked from at least COBOL and Algol, and something the average application program can identify with and easily understand.


The Thing I Love about
MCP Systems
7
Networking and
TCP/IP
Paradigm



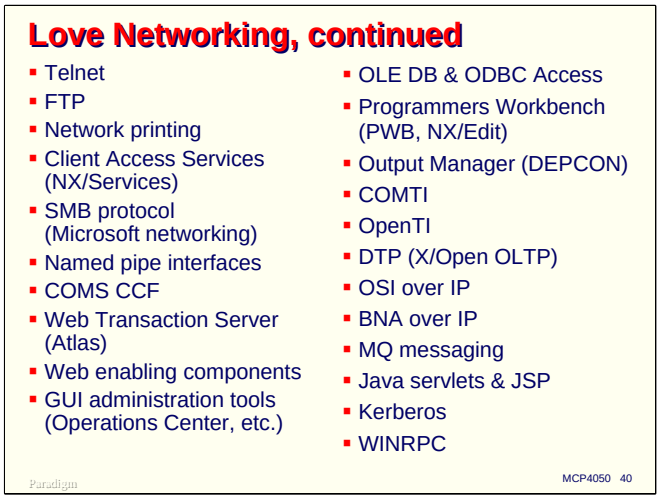
Another aspect of MCP systems that I have grown to love is networking, especially TCP/IP.

TCP/IP has been a growing force in MCP systems for about ten years now, ever since it was made independent from BNA and the CP2000. TCP/IP has completely revolutionized data communications for the MCP. TCP/IP has proven to be fast and reliable, It's also relatively easy to configure and administer, although the OI commands, especially those for configuration, are a little cryptic.

Combined with modern WAN facilities and the Internet, it has essentially removed distance as a factor from business transaction processing. I doubt that anyone wants to go back to the days of the "classic" datacom – particularly poll/select, two-wire direct interface (TDI), and synchronous modems.

TCP/IP has dramatically improved the major areas of communications for traditional business IT – terminal emulation, file transfer, and especially remote printing.

In addition to these traditional modes of communication, TCP/IP is the basis for most of the new features and functionality in the ClearPath line.



Just as an exercise, I tried to think of all the products and functionality we now have in ClearPath MCP systems which are either based entirely on TCP/IP or are at least heavily dependent on it. It is an impressive list, and I doubt that it's complete.

There are two particular areas of functionality which are enabled by TCP/IP that I specifically want to comment on.

The first is the SMB protocol (sometimes called "Microsoft networking"), which allows client workstations and servers – particularly Windows systems – to access MCP file, printer, and application resources. It has a companion product, the Redirector, which allows the MCP environment to access file directory and printer shares on SMB-capable servers – again, particularly Windows systems. Most shops seem to use this primarily for file transfer, but it also affords good capability for direct file access and integrating MCP printing with Windows print servers.

The SMB protocol was one of the original capabilities of ClearPath MCP systems, and I think in many minds is the one that really set ClearPath apart from A Series.

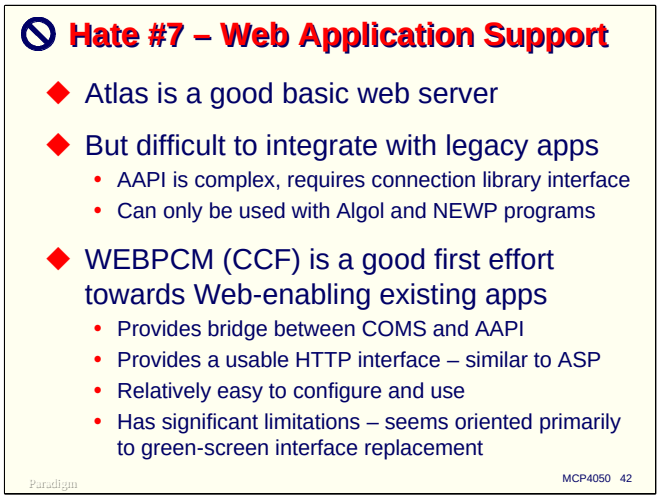
The other area, and to me one of the under-appreciated delights of the ClearPath line, is CCF – the COMS Custom Connect Facility. Basically, CCF provides a means to connect network resources to COMS so they act like COMS stations. This allows MCP programmers to use the standard COMS API to communicate with clients using TCP/IP ports (also known as "Berkeley sockets"), named pipes, and the web browser protocol, HTTP. It is extremely easy to configure and use. It offers efficient, high speed data transfer with low latency between clients and COMS programs.

The best part about CCF is how easy it makes it to interface TCP/IP ports to COMS applications. Programming to communicate directly with TCP/IP ports is done using either TCPIPNATIVESERVICE port files or the MCP Sockets Service. Both are rather complex interfaces, and require the programmer to deal with a lot of connection/disconnection, error handling, and state-management issues. CCF hides all of this complexity and just exchanges messages with the COMS programs.

The Thing I Hate about MCP Systems

Web Application Support

Paradigm



One of the primary roles the TCP/IP protocol plays in modern computing is as the underlying protocol for the World Wide Web. The Hyper Text Transfer Protocol (HTTP) used by web servers and web browsers typically runs on top of TCP/IP. Hyper Text Markup Language (HTML) is often the payload of an HTTP message.

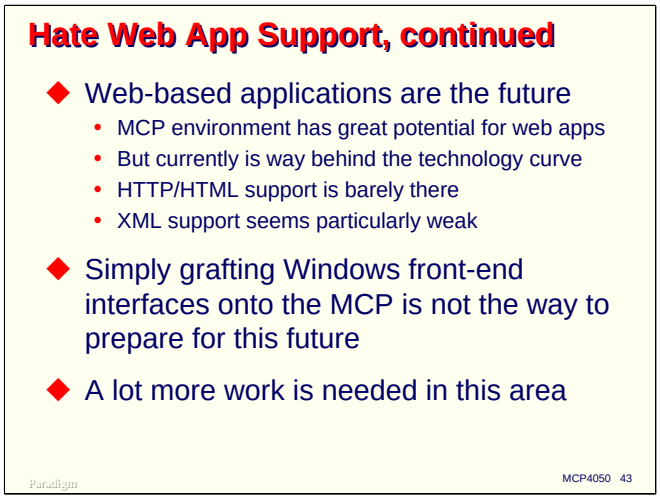
MCP systems have a native web server, Atlas – officially known as the Web Transaction Server. This is a good basic web server, with support for static content (e.g., HTML and image files), interfaces to MCP application programs, Java servlets, and Java Server Pages (JSP).

The main problem I have with Atlas is the manner in which it interfaces to applications. The primary interface, AAPI, is fairly complex, but worst of all, operates only as a connection library. This means that only Algol and NEWP programs can be used directly with Atlas.

In an attempt to address this, CCF was enhanced in MCP 5.0 with the WEBPCM. This component provides an interface for HTTP messages between AAPI and COMS. In conjunction with a utility library, WEBAPPSUPPORT, the WEBPCM allows any COMS application (particularly those written on COBOL) to receive messages from browsers and format responses to them.

I think the WEBPCM is a good first effort towards bringing web enablement to existing applications. It allows users to leverage existing COMS programming skills, and provides quite a nice interface to the complexities of HTTP messages. Once you master some basics of HTTP and HTML, the WEBPCM is relatively easy to configure and use.

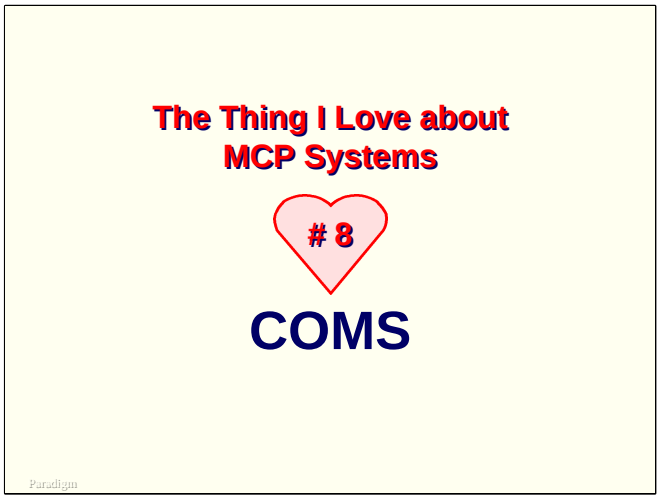
Even so, the WEBPCM has some significant limitations. It seems to be oriented primarily to handling small input messages and simple HTML responses – much as you would encounter with a straightforward replacement of 24x80 green-screen transactions. The WEBPCM becomes awkward and somewhat inefficient when you try to move very far beyond this very basic usage. All of the communications between the COMS program and Atlas must be channeled through the COMS message area, which has a small upper limit on its size – 64K bytes. This forces the programmer to be concerned with segmenting output, which is difficult, because the WEBPCM does not have any direct support for HTTP chunked messages. There is no support at all for incoming HTTP messages that are larger than about 50K. In many cases, these limits are too small to conveniently build an effective, modern user interface, which is one of the main points to using the web.

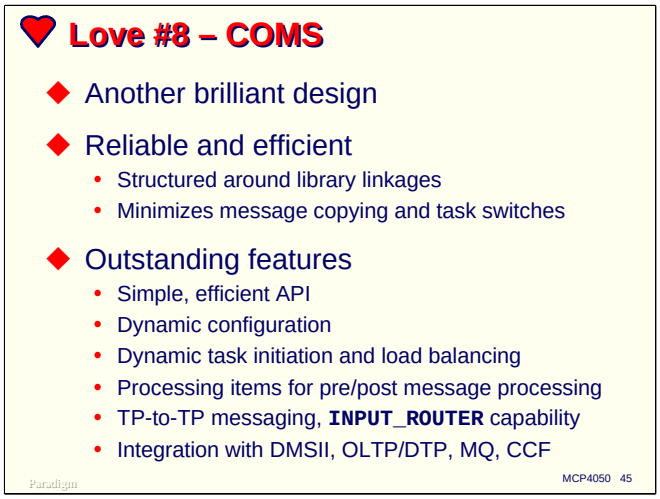


Web-based applications are a large part of the future of business IT, and MCP systems have great potential to be really good web servers and web application hosts. The problem is, they just don't have enough built-in support for such applications. You can build applications that use HTTP and HTML, but the tools are weak. Direct support for XML in MCP applications is particularly weak.

A great deal of work has been done in building front ends to MCP systems for web services – most of it using Windows systems. COMTI and SOMS are examples of this. Some of this front-ending is good, but it should not be relied on as a complete solution. Simply grafting Windows tools as front-ends to MCP applications is not the way to prepare for the future promised by web applications.

A lot more work is needed in this area for MCP systems to reach their full potential.


The Thing I Love about
MCP Systems
8
COMS
Paradigm



I have been using and programming for COMS since 1987, and have liked it tremendously from the beginning.

COMS is another brilliant design, and has matured well beyond its role as a generic Message Control System. Like many of the other strong elements of the MCP product line, it is both reliable and efficient. Its design is centered around library linkages, which contributes to its efficiency by minimizing the copying of messages from one buffer to another and switching processors among running tasks.

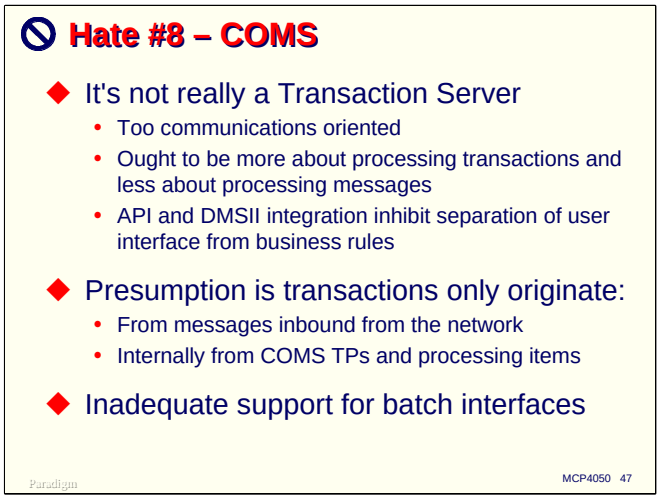
COMS has a number of outstanding features and capabilities, but the following are at the top of my list:

- **Simple, efficient API**. The Direct Window mechanism is quite powerful, but also quite easy to program for. The interface gives programs the ability to know quite a bit about the incoming messages and their environment, but programs can choose how much of this they want to have.
- **Dynamic configuration**. There is very little about the configuration of COMS programs, windows, and other entities that you cannot change while it is running. This is an outstanding feature, and we need to see more of this type of dynamic capability in other parts of the system. The ability to dump and load the COMS configuration to and from text files is also an extremely important feature.
- **Dynamic task initiation and load balancing**. COMS does a good job of controlling application tasks, starting and stopping them in response to current workloads. The tuning parameters for these are straightforward and can be changed on the fly.
- **Processing items**. The ability to write pre- and post-processing code for messages that is applied automatically outside the control of the application programs is one of the best ideas in the COMS design.
- **TP-to-TP messaging**. COMS programs can exchange messages with each other, and most importantly, can generate messages that get processed as if they had come in from the network by routing them to the built-in **`INPUT_ROUTER`**.
- **Integration with other tools**. COMS is integrated with several other middleware elements of MCP systems. Chief among these are:
  - **DMSII**. COMS synchronized recovery is a critical element in building reliable, recoverable transaction systems.
  - **OLTP/DTP**. The Distributed Transaction Processing interface to COMS simplifies the job of programming for the XA and XATMI protocols.
  - **MQ**. Similarly, the COMS interface to message queuing simplifies the job of programming and delivers the traffic to the place you need to have it most of the time anyway.
  - **CCF**. As previously discussed, CCF vastly simplifies the job of interfacing to TCP/IP ports, named pipes and the Atlas web server.

The Thing I Hate about MCP Systems

COMS

Paradigm

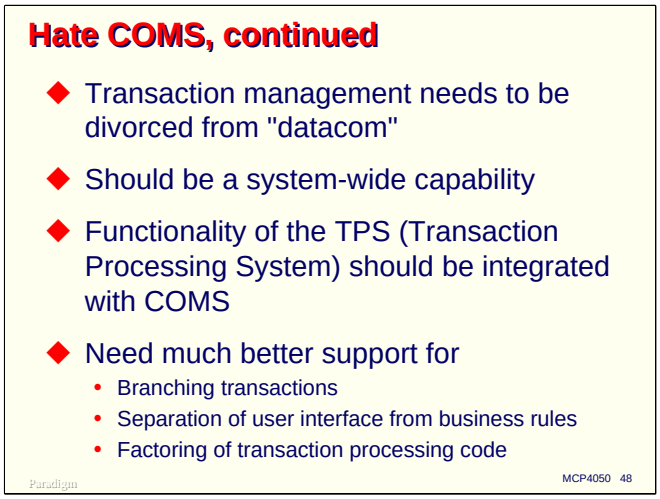


I don't really hate COMS, I just hate what it doesn't do.

COMS is now officially known as the Transaction Server for ClearPath MCP. One problem is that it's not really a transaction server. COMS is too rooted in its original role as a communications handler. Its focus should be more about processing transactions and less about processing network messages as if they were the only form of transaction.

Another problem is that the COMS and DMSII APIs inhibit both the separation of user interface issues from business rules, and factoring of business rule logic into functional and maintainable modules. The entire transaction must be processed by the COMS program itself. You can't even export the business rule logic to a library without abandoning synchronized recovery.

COMS operates as a closed system. It handles messages that come from the network and its own internal programs, and from nowhere else. There is no practical way for, say, a batch program running from a WFL job to route transactions to COMS and have them processed.

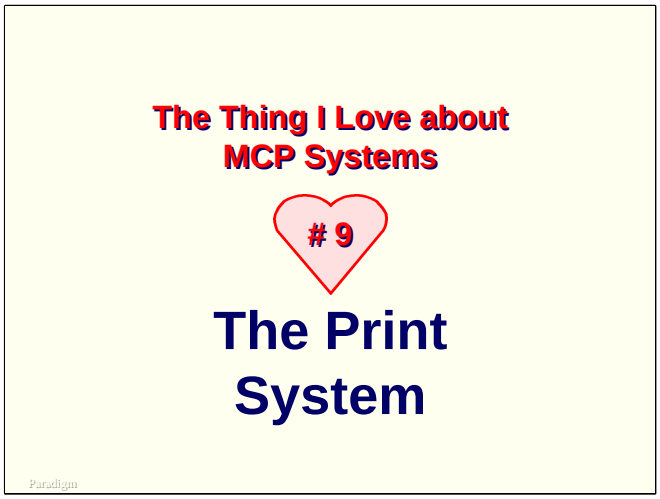


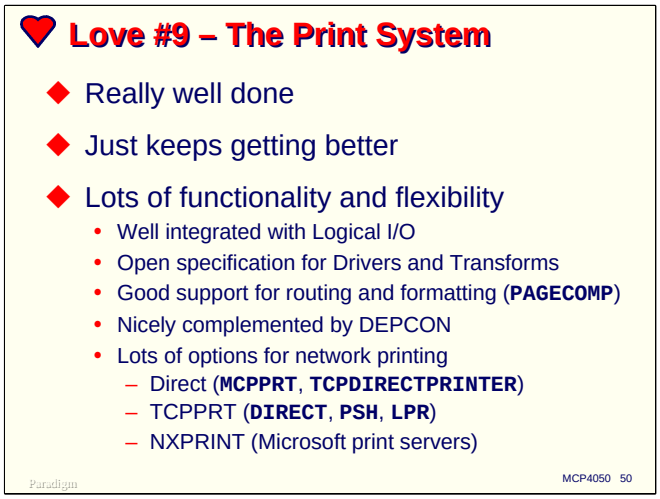
To truly be a Transaction Server, the management of transactions needs to be divorced from messaging and communications and made a facility that is available system wide.

The Transaction Processing System (TPS), a little-known and under-appreciated companion to DMSII, provides capabilities more in line with what is needed in this area. One idea would be to bundle the TPS and integrate its transaction routing, synchronized recovery, and journaling capabilities with the communications handling and routing capabilities of COMS.

We need much better support in MCP systems for some related aspects of transaction processing:

- **Branching transactions**. Transactions often beget other transactions. We need to be able to handle this in a safe, coordinated, and recoverable way. Since it's not practical to implement every transaction in one big program, this means the system must support the concept of a single data base transaction spanning multiple software modules. One way to do this is with OLTP/DTP, but that is an expensive solution for transactions that are internal to a system.
- **Separation of user interface from business rules**. One of the really frustrating parts of designing on-line applications for the MCP is the difficulty of separating editing and formatting concerns from the actual processing of transactions. Not being able to do this leads to the same transaction being implemented in many places throughout the system, which in turn leads to problems in keeping the various instances of code all in synchrony. This is a problem that grows geometrically with the size of the application.
- **Factoring of internal transaction processing (business rule) code**. Once you succeed in separating the user interface portions of an application from the internal transaction processing or business rule portions, it is still highly desirable to be able to factor the business rule logic into separate modules so they can be implemented and maintained individually. This helps reduce the overall complexity of an application's design, and the difficulty with maintaining it over time. It is also closely associated with the ability to implement branching transactions.


The Thing I Love about
MCP Systems
9
The Print
System
Paradigm



One of the strongest areas of the MCP is the Print System. Printing for the MCP has come a long way since the pre-Mark 3.6 Autobackup days. It is just really well done, and keeps getting better with every release.

The primary thing I like about the Print System is that it has lots of functionality and is very flexible. It's well-integrated with Logical I/O, and is driven primarily by file and task attributes. The PRINTDEFAULTS mechanism is a particularly good idea.

Another fine aspect of the Print System design is its openness. It supports standard PCL and Postscript drivers in addition to the customized ones for Unisys-branded printers. You can also write your own drivers, transforms, virtual servers, and I/O handlers to meet special needs. The operational interface is also open, and accessible programmatically through the **DCKEYIN**, **ASERIES_INFO**, **PS_COMMAND**, and **PS_SET** APIs. This openness has led to a market for third-party tools, such as those from GoldEye Software.

The Print System has good facilities for routing and formatting print streams. I particularly like the PAGECOMP attribute, which provides a device-independent mechanism for specifying page composition parameters to the device drivers.

The functionality of the Print System is further enhanced by the Unisys Output Manager, more familiarly known as DEPCON. This combination, along with the various add-ons to DEPCON, such as DDA, can provide a whole additional dimension of formatting and routing capability.

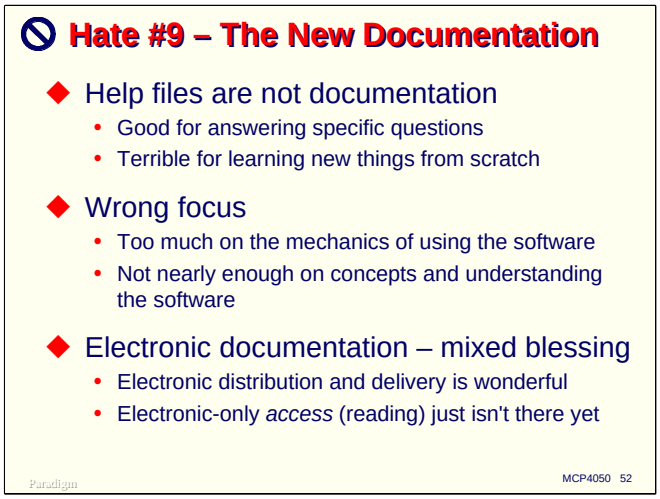
One of the things I like best about the Print System is its support for network printing. In addition to routing print streams through DEPCON for network delivery, the Print System can connect to direct network printers (such as HP JetDirect devices), Microsoft Windows print servers, and Unix PSH and LPR interfaces. MCPPRT is an excellent replacement for traditional terminal-based pass-through printing, and has the advantage that the print traffic does not need to be routed through COMS.

The Print System is one area where I just do not have any substantive criticism.

The Thing I Hate about MCP Systems

The New Documentation

Paradigm



An area of MCP systems that deeply concerns me, both in terms of its current state and its apparent direction, is documentation. The object of my concern is the current fascination with help files.

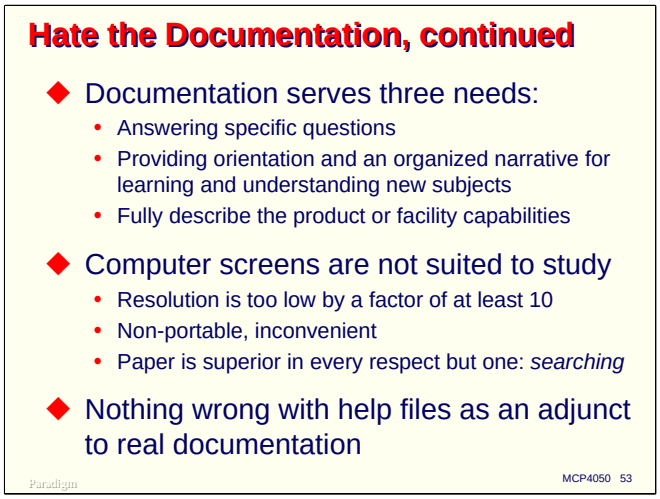
Help files are not documentation. The purpose of help files is to, well, give you help. They are good for answering questions when you already know quite a bit about the subject, but terrible when you are trying to learn a new subject or product from scratch.

The main problem with trying to use help files as a substitute for documentation is that they have the wrong focus – they are too concerned with the mechanics of using the software. They do not focus on the things you really need when trying to study and learn something new – overall concepts and understanding.

WinHelp documents are the worst of the new documentation formats. Of these, I think that the CCF documentation is the worst of the worst, with the material for Client Access Services a close second. These "documents" have a very poor approach to the organization and sequencing of information. The topics are too short and choppy (almost as if Dykstra had once written a paper, "Scrolling Considered Harmful"). They often illuminate the obvious while ignoring the important (the CCF syntax diagrams are a prime example of this). Finally, the WinHelp files are just about worthless when printed – you get all the words, but in a big pile of paper that has no organization whatsoever.

The newer HTML Help is generally better than WinHelp, but not as good as a well-designed linear document. I think that the PDF documents are generally well done, and offer a good compromise between random access to answer questions and a linear organization to promote study and learning.

Electronic documentation in general is a mixed blessing. The ability to electronically distribute and deliver documentation – on CD-ROMS, over the Internet, whatever – is simply wonderful. It makes it possible to always have complete, up-to-date documentation in a compact, portable, and persistent form. Electronic access to printed material is still primitive and simply not ready to completely take over either paper documents or the the traditional organization of information that has been developed for paper documents.



Documentation must serve three main needs:

- It has to be able to answer specific questions about the subject that it documents. This is the only area where help file technology is successful.
- It has to provide orientation and an organized narrative for the new user to assist them in learning and understanding new subjects. Most help files fail miserably at this. If they didn't, there would not be several hundred feet of shelf space in the computer section at every Borders and Barnes & Noble.
- It has to fully describe the capabilities of the product or facility it documents. Too many of the help files for MCP systems currently fall short of this requirement.

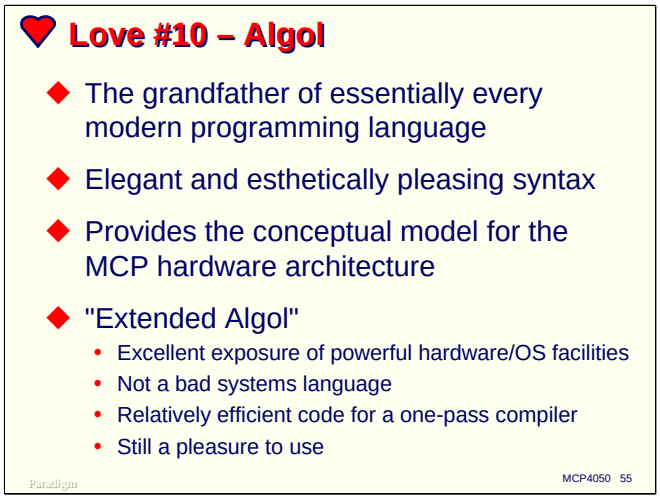
Another problem with "on-line" documentation is that computer screens are not very good for extended reading and study. The resolution of current screen technology is lower than most printed documents by at least a factor of ten.

While electronic documentation itself is very portable, *access* to electronic documentation is not. It's hard to take a chapter with you to read at lunch, unless you take your laptop along with you.

Several studies over the years have shown that for reference and information transfer, paper is superior to electronic presentation in every respect but one – searching.

Please understand that I am not arguing for the elimination of help files. They have their place as an adjunct to traditional documentation. They are not a substitute for it.


The Thing I Love about
MCP Systems
10
Algol
Paradigm



The Algol programming language is the first thing I loved about MCP systems, starting with the B5500 in 1966. It is still one of the things I love most about MCP systems today.

Algol is the grandfather of essentially every modern programming language – including Pascal, Ada, Python – even C, C++, and Java. I still find its syntax esthetically pleasing, satisfying to write, and easy to read.

As I mentioned earlier, Algol provides the conceptual model for the MCP hardware architecture. The Burroughs B5000 and B5500 were originally developed to support Algol. Their design was considerably refined to support even richer Algol usage for the B6500, the direct ancestor of the ClearPath MCP systems we use today.

Algol was originally designed as a reference language – a notation for publishing algorithms. The Algol 60 report lacked mention of basic practicalities such as I/O.

Burroughs took this raw base and transformed it into "Extended Algol," in the process turning it into a real programming language. The current implementation does an excellent job of exposing some powerful aspects of the hardware and operating system in a safe manner – including string manipulation, bit manipulation, I/O, fault handling, inter-process communications (including libraries), and pointers and reference variables.

By exposing a number of APIs in the MCP, Algol has become a good systems language, and we often use it much the same way that assembly language is used on other systems – to create wrappers around system facilities so they can be used by other languages (particularly COBOL).

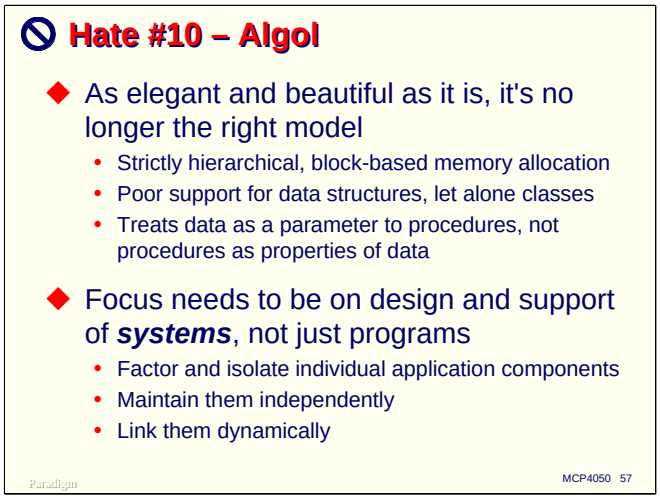
Algol is still a fast, one-pass compiler, and generates relatively efficient object code considering this.

All in all, I still find Algol on MCP systems a pleasure to use.

The Thing I Hate about MCP Systems

Algol

Paradigm



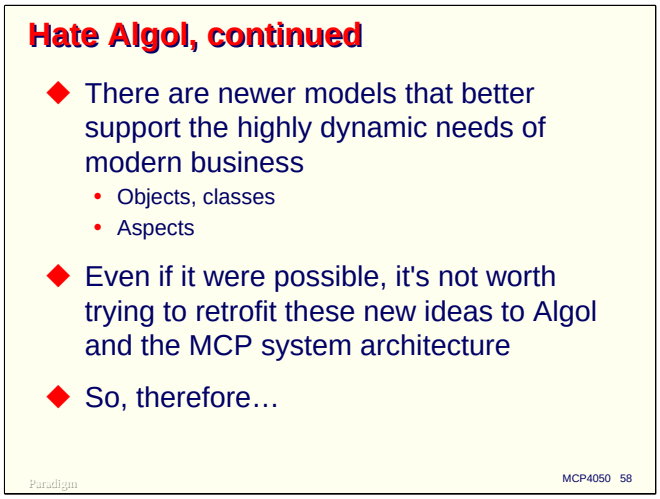
As much as I still love Algol, and as elegant as its coding can be, it is not the right model for generating applications or providing the foundation of a modern system architecture.

The complaints I have concerning Algol largely echo those that I have about the MCP system architecture – not surprising since they are so closely coupled.

- Its strictly hierarchical, block-based approach to memory allocation.
- Its poor support for data structures and object classes (despite the capabilities of "defines" and structure blocks).
- Its overall focus on the primacy of code over that of data. Essentially Algol treats data as a parameter to procedural code. What we need is the inverse of this – code should be associated with the data and treated as a property of the data.

Algol may be a good language for writing programs, but what we need are languages that are good for writing *systems* of programs. In dealing with systems, the principal problems are managing both modularity and the coupling between the modules. These problems grow geometrically with the size of the system. We need to be able to factor and isolate individual application components, maintain them independently, and link them dynamically as needed.

MCP systems have some wonderful capabilities that address these needs, but they simply do not go far enough. More significantly, I do not think they can go any farther.



Algol has served us well over the past 40 years, both as a vehicle for programming and as a conceptual basis for the MCP system architecture. Computing has changed dramatically over those 40 years, however – both in terms of hardware and our understanding of how to construct and maintain software systems. There are now much better models upon which to base a system architecture. The most promising ones that I know of involve objects, classes, and the relatively new concept of aspects.

It is a common misconception that hardware technology has advanced much faster than software technology and that software is not able to use current hardware facilities to the fullest. Quite the opposite it true. Hardware has merely gotten a lot faster and a lot cheaper. There have been no significant advances in hardware architecture in at least the last 30 years.

Even if it were somehow possible, I do not think it is even worth trying to retrofit modern programming and software concepts to either the Algol language or the MCP system architecture.

Given this, I have reluctantly concluded that …

Algol must go.

I do not mean by this that Algol as a programming language should be banished, or all Algol programs should be eliminated. I look forward to continuing to use Algol and program with it for the rest of my career.

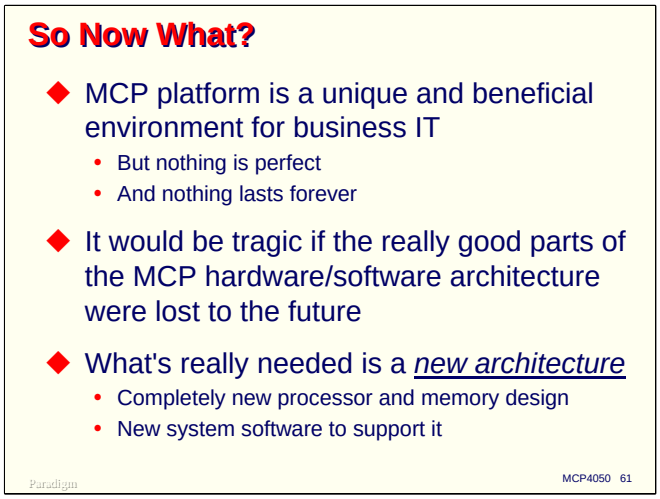
Rather, I mean that Algol can no longer be the basis for the flagship Unisys computer system products. Its day has passed, and it is time for it to enter retirement.

More importantly, since I think the conceptual basis for MCP systems is no longer adequate, I have also reluctantly concluded that …

There is no long-term future for the existing MCP platform.

This is not to say that MCP systems as we know them will (or should) go away any time soon. I think they will continue to be viable products, and should be actively enhanced and marketed by Unisys, for at least ten more years. That remaining life, however, will be spent primarily in support of existing customers and applications. I do not believe that much, if any, significant new application development for the platform will take place from this point forward. This means that there will no significant growth in use of this product line.

We have been seeing a slow drain of existing customers and applications from the platform for some years. This will continue, probably at something close to its current pace, with very little in the way of new customers and workloads to replace it.

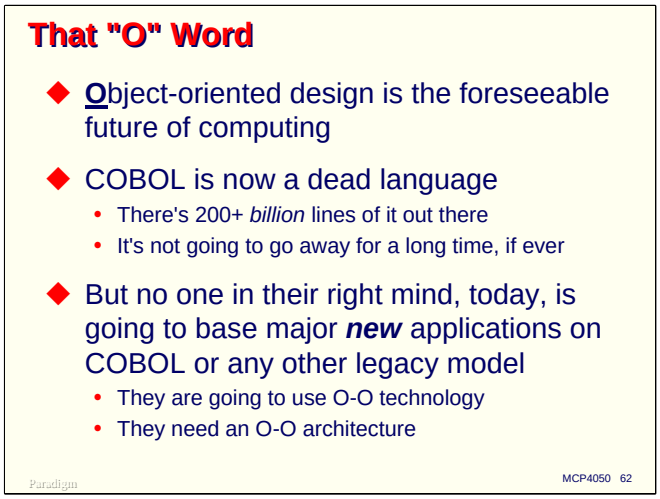


Given that the existing MCP systems and their architecture are doomed to a slow but inevitable death, what do I think should be done?

First of all, I think that should be allowed to happen. The MCP platform is a unique and beneficial environment for business IT applications. Nothing is perfect, however, and nothing lasts forever. The really good parts of the architecture need to be carried forward and reinvented in a more modern context.

It would be tragic if these good parts – especially the benefits of descriptors and tags; the separation of code, data, and control information; structured and controlled access to memory addresses; and automatic process state saving – were lost to the future.

What is really needed to carry these good things forward is a *new architecture*. This means a completely new processor design, probably a new memory design, and new system software to support the new hardware.



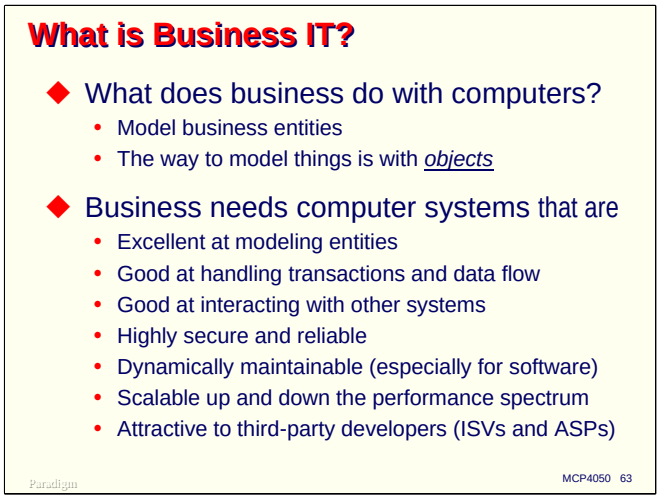
You have probably noticed me making liberal use of that "O" word – *objects* – and the "O-O" word – *object-oriented*. I have done this intentionally because I think that objects, classes, and their related mechanisms are the foreseeable future of computing, especially in the area of application software design. If this is the future, then any new architecture needs to support it. More than that, I think object orientation needs to be the central theme of any new architecture.

I make the proposition that COBOL is now a dead language. Even with the O-O extensions in COBOL 97 and 2002, I seriously doubt that major new application development – especially by third-party software developers – is going to take place in COBOL.

An unavoidable issue with COBOL, however, is that there is so much of it out there running as part of enterprise-critical applications. One statistic I have came across, from Ronald Smith of Unisys (who was quoting a Gartner survey) is that there are at least 200 *billion* lines of running COBOL in the world, with some estimates ranging up to 5 *trillion* lines. This number continues to grow each year, but my guess is that most of it is revision and enhancement of existing systems, not really new work. In any case, COBOL is not going to go away for a long time, if ever.

COBOL programmers *are* going away, however, and are not being replaced at anywhere near the same rate they are disappearing. This is more probably predictive of the future of new COBOL development than anything else.

I do not think that anyone in their right mind, today, is going to build major new applications from scratch in COBOL. They are going to use object-oriented technology. For that they really need an object-oriented architecture to support them.



If you think about it, what do we do in business IT? Basically, we model business entities – customers, products, accounts – just about everything about the business. The best way we currently have to model things in software is with objects.

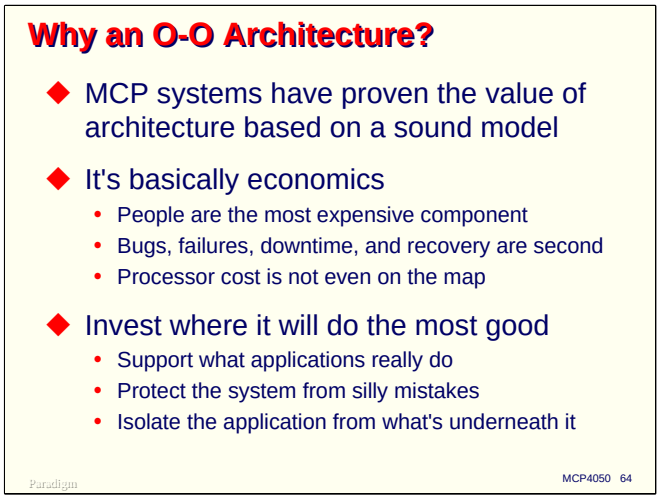
This suggests that what business needs are computer systems that are really good at modeling things and maintaining those models. Some software approaches are good at this, but the computer systems themselves are not. This is an opportunity.

In addition to modeling, computer systems for business IT also need to be good in a number of other areas:

- Handling transactions and data flow.
- Communicating or otherwise interacting with other systems.
- Highly secure and reliable.
- Dynamically maintainable, especially in terms of software configuration.
- Scalable up and down the performance spectrum.
- Attractive to third-party software developers, primarily Independent Software Vendors (ISVs) and Application Service Providers (ASPs).

All of these except the last are areas where traditional mainframe systems excel. I think more work needs to be done in the area of dynamic maintainability, especially concerning the process of updating software versions, to support the increasingly 7/24 nature of business computing. Computer systems for business must be attractive to third-party developers, since that is where most of the new application development is going to come from.

Microsoft understands this last item very well. It is the cornerstone of much of their success.



There is lots of object-oriented capability in the marketplace already, so why do we need computer systems with an architecture specifically suited to object orientation?

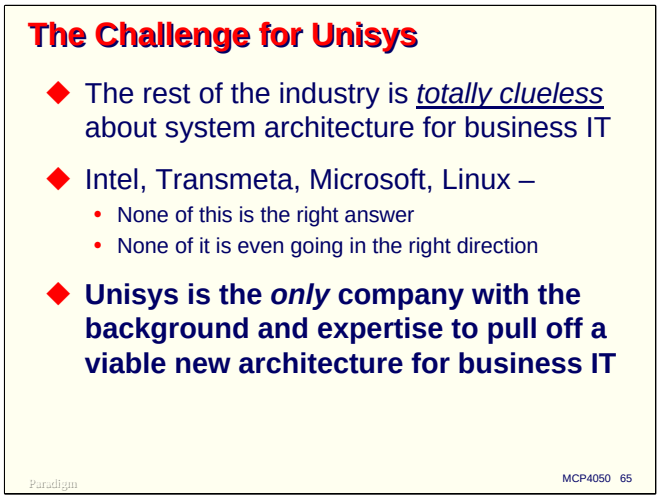
There are two main arguments for this. The first is that MCP systems have proven the value of an architecture based on a sound conceptual model. Instead of trying to do everything in software on top of a relatively primitive hardware base, responsibility for performing functions and protecting against boundary and operation violations should be layered. Some of the layers should be implemented in hardware, some in software, and perhaps some in layers between.

Just what these layers should be and how the responsibilities should be distributed among them is a rich field for study and debate, but it is clear to me that things like bounds protection and being able to distinguish among data, code, and control information need to be firmly embedded at the lowest level of the architecture. We see the benefits of tags and descriptors in the MCP architecture. Those benefits simply would not accrue if they were not universally and continuously enforced at the hardware level.

The second argument is basically an economic one. People are by far the most expensive component of computing. The cost of dealing with bugs, failures, downtime, and recovery from all these is the second most expensive component. The cost of processors in a system is not even on the map. Yet, as an industry, we continue to insist on economizing on this least expensive resource, and glorying in processor performance over operational safety.

We need to invest in computer system architecture where it will do the most good. The architecture should support what applications really do. It should also protect the system and individual applications from silly mistakes. Programmers are going to continue to code the potential for boundary violations and improper operand usage into their programs.

From reports in the trade press over the last year, it appears that Microsoft thinks it can browbeat their programmers into avoiding such problems simply by being more careful. Forget it. These kinds of problems are too sensitive to data dependencies for us to ever hope to eliminate them by inspection or testing. They will always be with us, and will always be a potential source of unauthorized intrusion. The only way to prevent such problems from causing damage is to isolate the applications from the raw hardware capabilities and enforce the rules at levels below the application. That means taking a more sophisticated approach to designing those lower levels of the architecture that support applications.

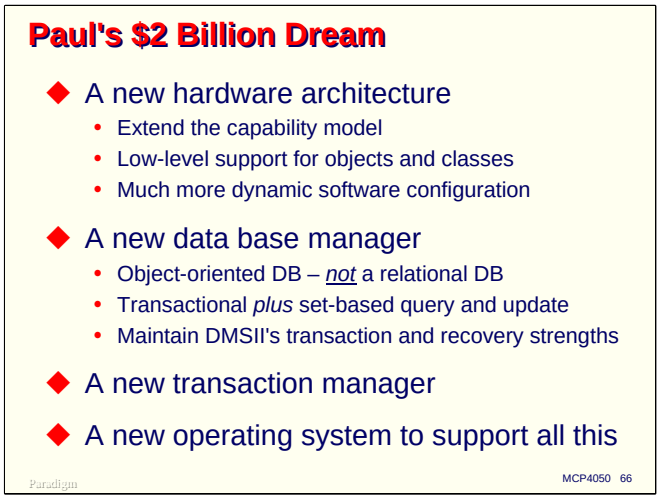


Here is a challenge I present to Unisys.

The rest of the industry is clueless, and I mean *totally clueless*, about system architectures for business IT. IBM understands the needs of business very well, and they, Sun, Oracle, and Microsoft are working hard to build software foundations to support business IT, but no one is looking beyond software-based solutions to the whole picture of system architecture.

None of the existing computer system, processor, or operating system architectures in existence today – not Intel, not Transmeta, not Sun, not Microsoft, not Linux – none of them – are the right answer. Further, none of them (with the possible exception of the IBM AS/400) are even going in the right direction.

Because of its experience over the past 40 years with MCP systems, I think that Unisys is the ***only*** company in the industry with the philosophical background and technical expertise to design and build a viable new architecture for business IT. Whether they can pull it off is an open question; whether they will even try is an entirely different question; but to my mind no one else in the industry today even knows where to start.



I have a dream.

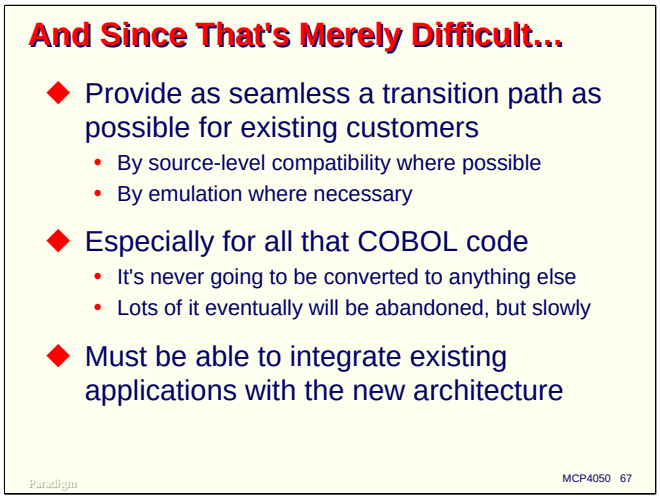
I have a dream that the existing MCP system architecture can be reinvented to support modern software design concepts and to carry forward the really good parts of the existing architecture. I would like to see the capability model extended to provide greater protection to the system and applications, and to eliminate the need for trusted compilers. I would like the concepts of tags and descriptors carried forward to provide support in the lower levels of the architecture for objects and classes, especially in the areas of method binding, memory allocation, memory addressing, and garbage collection. I would like to see much more dynamic software configuration. The goal should be that the system needs to stop for nothing, not even operating system upgrades. This may not be completely realizable in practice, but that should be the standard against which architectures are judged.

I have a dream that the reliability and recoverability of DMSII can be carried forward into a new data base management system. I would like to see this new DBMS have many of the beneficial query and update characteristics of a relational data base system, but it should not be limited to the relational model. I would like to see it be an object-oriented DBMS to match the real needs of business IT and to exploit the underlying capabilities of the new architecture.

I have a dream that the new architecture would support a new transaction manager, one that really understands what transactions are, and one which facilitates factoring of applications into functional and maintainable units.

Finally, I have a dream that we can have a new operating system to support all of this. Many of the good concepts of the MCP can be carried forward, but to support such a new architecture and the data management and transaction facilities, I think it needs to be a design redone from the beginning.

This is, admittedly, quite a dream. I think of it as my two billion dollar dream, because that is what it will probably cost, plus or minus an order of magnitude, to realize it. That is a lot of money, but that amount of money is probably going to be spent just on MCP system engineering over the next ten years anyway. The question is what Unisys and its customers will get in return for that investment.

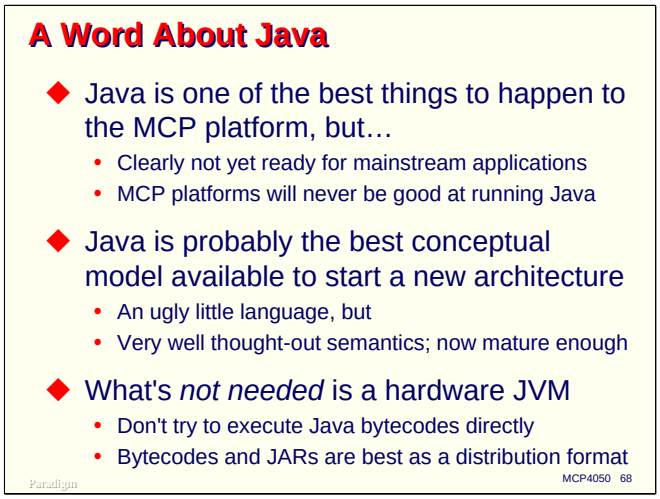


That is not the end of my dream.

There must be a transition path for existing customers from the existing MCP system architecture to any new architecture that is developed. Ideally, this transition can be accomplished by source-level compatibility. That may not be entirely practical, so at least some emulation of the existing architecture on the new one may be necessary. As Unisys has shown with their LX and CS product lines, they are good at emulating the existing architecture, so I think that goal can be reached.

This transition path is especially important for COBOL, as that makes up the vast majority of most customers' code bases. All that COBOL code is never going to be converted to something else – lots of it will eventually be abandoned as the applications are reengineered and completely redeveloped using other languages and environments, but overall the abandonment will be a slow process.

Finally, I do not think it is sufficient just to have a separate environment, emulated or otherwise, for existing applications. The transition between old and new will be very gradual – spanning years, perhaps decades – and it must be possible to integrate existing application into the new architecture piecemeal.

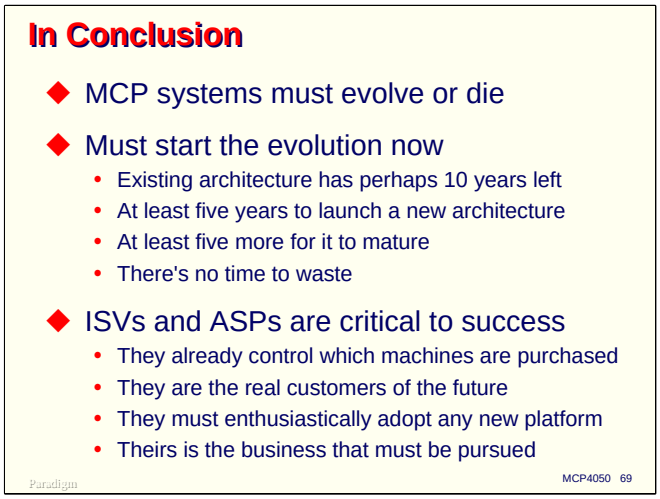


I think Java should have a role in guiding the design of any new architecture to replace MCP systems. Java is potentially one of the best things to happen to the MCP platform, and I am very pleased and encouraged by the interest Unisys has shown over the past year in getting serious about Java.

The current implementation of Java on the MCP is clearly not yet ready for mainstream applications. I think this situation can be improved quite a bit, but I doubt that the existing architecture will ever be very good at running Java. A lot of the ideas in Java resonate very strongly with the architecture and philosophy of MCP systems, but Java and MCP systems are on very separate, if quite parallel, tracks. Each one has good reasons for being on their particular track, but I do not think the fundamental differences between them can ever be satisfactorily resolved. This is just one more reason why MCP system architecture needs to be completely rethought and redesigned.

Whatever the ultimate success of Java in business IT, I think it is probably the best conceptual model from which to start thinking about the requirements and design of a new architecture. From a coding perspective, I think Java is an ugly little language – too similar to the most over-rated language of all time, C. I can understand why the originators of Java chose the C syntactic model. Regardless, they gave (and continue to give) a great deal of thought to system issues. As a result, I think that Java has the most completely thought-out semantics of any software environment available today. After ten years of development and use, I now think that Java is now mature enough to be used as the starting point for a new system architecture.

Note that I am proposing that Java be considered a starting point. What we do *not* need is a hardware-based Java Virtual Machine (JVM). The goal should not be to execute Java bytecode programs directly. Bytecode files and JARs are best thought of as a reference model and software distribution format. The new architecture should be able to translate these into its native execution format, not execute them directly.



In conclusion, I propose to you that MCP systems must evolve into something entirely different than they are today, or die. That death will not come soon, but the current architecture is definitely doomed to a long, slow slide into oblivion. Their evolution must be something much more radical than anything we have seen in the last 40 years.

The process of evolution needs to start now. I think that MCP systems have perhaps ten years of life left. I think it will take at least five years to do the conceptual and engineering work to get a new architecture launched. It will take at least five more years for such a new architecture to mature to the point where it can completely replace the existing product line. Five and five is ten. There is no time to waste.

Finally, I want to emphasize once again the importance of attracting third-part software developers to a platform. The days of customers choosing computer systems themselves is drawing to a close. More and more, customers are choosing a software package and then buying the platform that package runs on. In that sense, ISVs and ASPs already control which machines are purchased. They are the real customers of the future. In order for any new platform to succeed, it must be enthusiastically adopted by the third-party developers, so theirs is the business that must be pursued.

Copies of this presentation may be obtained at http://www.digm.com/Unite/2004.