DOI:10.1145/3133319

Article development led by acmqueue
queue.acm.org

The network era requires new models, with interactions instead of algorithms.

BY ANTONY ALAPPATT

Network Applications Are Interactive

THE PROLIFERATION OF mobile devices and the interconnectivity between them has created new application opportunities. These new applications are no longer limited to a single system space but are spread across many system spaces. This shift of application from single system-spaced, host-based systems to multisystem-spaced solutions is being hampered by software toolsets that are stuck in the older sequential computational models.

This article defines this new requirement as a network application. It argues how current programming based on computation is not the right model and suggests that application development has to move from software based on a computational premise to a communication or interaction premise. It closes with a simple CRUD (create, read, update, delete)-based application developed with interactions to illustrate the point, laying out the challenges and explaining how such a system will not have algorithms at its center but will work by using interactions.

Imagine a patient-doctor encounter in the future, where the patient, who has a personal electronic medical records (EMR) device, is identified to the doctor's office either by an near-field communication (NFC) wave or a physical scanning of the EMR device into a receptacle. The EMR device talks to the doctor's office system, providing all the necessary information. Prescriptions could be sent directly to the pharmacy, with a sync point given to the EMR. Thus, when the patient goes to the pharmacy, the medicine is delivered promptly.

This is the network era, requiring new approaches to software. Host-centric programming that limits itself to single system space will not be sufficient for the multisystem-space needs of this new digital era.

What Is a Network-Based Application?

Applications are of two types: stand-alone and networked.[2,9] Applications such as word processing on PCs, multiuser programming on a mainframe, or distributed computing using Simple Object Access Protocol (SOAP) or object request broker (ORB) are stand-alone. Stand-alone applications provide a single machine view[9] to the application programmer by having a single entry point (that is, the calling program hands over control to the application and takes back control after the application is complete). The application accesses data from the environment by executing I/O routines that runs parallel to the main routine. The program counter in the machine controls the program. The stand-alone application runs on top of the machine (virtual or real).

In the example EMR application, notice how the programmer does not view the application as one hierarchical pro-

December
18
sign

gram but as interactions by different people that take place across a network of devices. Unlike a stand-alone application, a network application advances when the different agents interact with each other through their actions. These actions accept application state changes from agents on the network and also affect application state changes on other agents on the network. The network-based application runs on top of the network infrastructure.

Network Application Is Not Computation

The modern digital era requires people and devices all acting in unison to create the overall business experience. All these different computational agents sitting on different system spaces should communicate with each other. To put it simply, they should ask and tell each other to create the whole solution. This digital-era choreography of events happens simultaneously and, hence, is a concurrent problem.

Programming consists of statements separated by a sequential operator (as in Java) or an assignment operator. These operators assign the value of the expressions into a memory location and also instruct the compiler to move to the next instruction. These two fundamental constructs of the language make programming very sequential. Programming in a concurrent world, with sequential constraints, places a big challenge on the programmer.

Generally, there are two main parts that make a program work. One is the control of the program, and the other is what data is transferred when the control moves forward. The control is the cursor where the program is operating. In sequential programming, the control moves from top to bottom. The only ways to alter program control are by using go-to statements; exception statements; and iteration/looping statements. Programming consists of preparing the data so the processor can process the information. The creative part of software is established by manipulating program control. Manipulating hierarchical programs to represent all sorts of models is difficult to achieve and, hence, leads to bugs.

In sequential programs, program control is assumed to move forward. The language itself does not have any facilities to handle processing across system spaces. If part of the execution is sitting in another system space, then how is the issue of control handled? How will the language have enough concepts to handle the different issues of multisystem-space computing such as (a) transferring control; (b) handling latency; and (c) dealing with exceptions? How will one piece of code tell another piece of code sitting in a different system space that it has continued successfully or has thrown an exception. Nothing in these languages supports such issues. They assume there is one system space and that the control in the processor will move forward to the next step.

Digital-era solutions must coordinate people and devices as peer-to-peer collaborating systems. Creating such collaborating systems from sequential programs is a big challenge. Is there a better way so that these concurrent systems can be expressed more directly to make digital solutions easier? Toward this end, the basic default sequential control of programs needs to be eliminated. The default sequential control makes expressing concurrent problems more difficult, as the programmer has to manipulate the sequential control to create concurrency.

One of the biggest issues is sharing state across system spaces. In the current programming paradigm, state is examined using functions and variables. These are available only within the constraints of the language and are not exposed outside of the operating language. To overcome this, the language itself should have facilities to share state (both program control and data) so that the client can examine it.

In some situations the application is aware of the location of the data but is not sure when all the data arrives. This makes the current deterministic nature of programs unsuitable. This nondeterministic nature of data in the digital era contributes to the inability of current programming to solve digital-era problems. Although fundamental programming does not have a way of dealing with nondeterministic systems, it is now achieved by following a new programming paradigm called reactive programming.[3,8]

Once state and control of a program are shared, is it possible to ensure that only authorized people have access to both of these crucial elements? Current programming languages do not have the concept of hiding information. Ideally, programming languages should have facilities

Figure 1. Computation is communication.

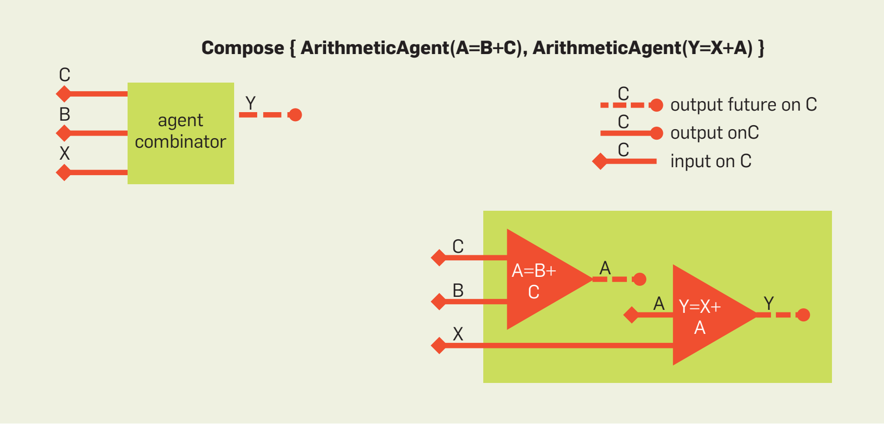


Figure 2. “Hello, World” application.

`#include <stdio.h>`	include information about standard library
`main()`	define a function named main that receives no argument values
`{`	statements of main are enclosed in braces

`    printf("hello, world\n");`
`}`
	main calls library function `printf` to print this sequence of characters; `\n` represents the newline character

to share state and provide controlled access to this selectively.

A network application that can work across system spaces needs a new way of looking at the computation problem. Rather than having *computation* ideas (procedure, functions, among others) as fundamental, it should be the *communication* ideas that are fundamental for a good network application programming paradigm.

Computation Is Communication

Consider two arithmetic expressions: Y=X+A and A=B+C. With the sequential programs now in use, these are written sequentially as { A=B+C; Y=X+A }. The programs operate sequentially to get the answer. If they are accidently written as { Y=X+A; A=B+C }, the program will continue to operate, but the answer will be wrong (i.e., a bug). Is there a way to eliminate this anomaly?

Imagine it a bit differently. In the same computation, the expressions run concurrently and combine when A is shared—that is, the composition consists of two agents, A=B+C and Y=X+A, which operate concurrently (Figure 1). The right-hand side of an arithmetic expression is a *sink* of the value and the left-hand side of the expression is a source of information. So in the case of A=B+C, B and C are sinks and A is the source. A=B+C is resolved when B and C arrive and raise A. Meanwhile, Y=X+A is waiting on X and A. Assuming X has arrived, when the first computation raises A, it is consumed automatically by the second computation to raise the answer Y.

Notice in this case the agents run concurrently, but because the second agent is waiting on A, that is raised by the first agent. The two computations are concurrent. The two agents are combined but they expose the combination actions (i.e., they accept C, B, and X and produce Y). This eliminates the propensity of bugs caused by the out-of-order coding in sequential languages.

The whole computation is running in parallel and drives itself, depending on the arrival of the values. The computation is no longer about algorithms but is about communicating values between two active computation agents. The flow of data is accomplished by naming the two variables.

Digital-era solutions must coordinate people and devices as peer-to-peer collaborating systems. Creating such collaborating systems from sequential programs is a big challenge.

Thus, to flow Y to another agent, all that has to be done is to combine AgentCombinator with the arithmetic agent that is going to consume Y. To define this communication, the control is established by splitting the communication into two pieces: one representing the *giving* of information and the other representing the *receiving* of information.

Making the computation concurrent addresses the speed of the computation, eliminates the issue of bugs when the computation is coded out of order, and makes the computation networked and nondeterministic. Clearly, computation is better expressed as communication.

Network Application Is Communication

Unlike a stand-alone application, a network application does not have a single machine view. The network application is a swarm of many machines operating concurrently, also known as agents. These agents, residing in single or multiple system spaces, coordinate their work to create an experience. Each agent could be acting as a client in one instance and a server in another.

Whether stand-alone or networked, a coherent application has certain salient properties.

Application control. It is the application control that sets the rhythm of the program. Once the program is given control from the operating system (through the `main()` function in C) or from a Web server in the case of a service, the programmer manages program control by applying different control statements provided by the employed language. After the program is completed, control is handed over to the operating system or the service initiator.

The magic of software is accomplished by controlling how these instructions are organized according to the language rules. For the sake of illustration, consider the C language.[4] The only control elements that are given to the programmer are statements of functions, assignments, and looping statements. The programmer uses these constructs to build the application flow (Figure 2).

Once a stand-alone application in C has received control, the program

Figure 3. "Hello, World" application as functions.

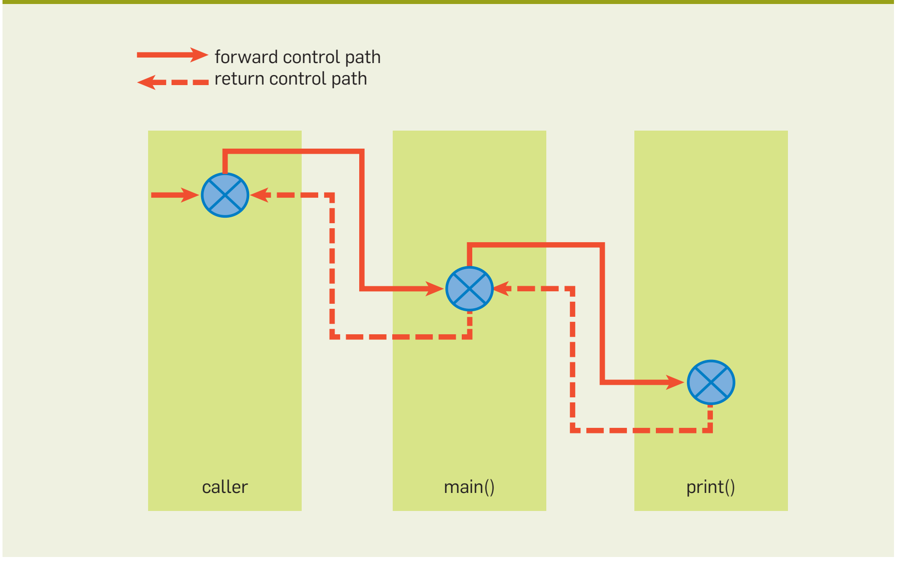
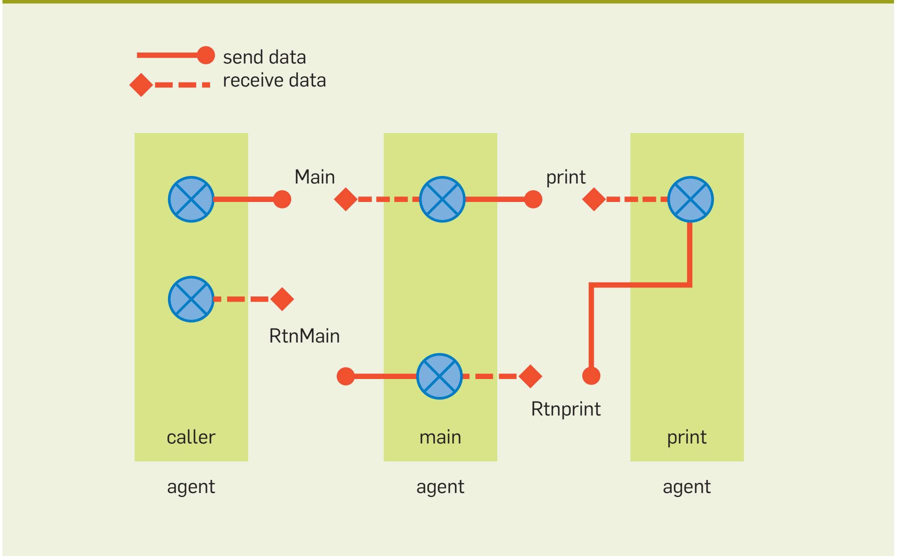


Figure 4. "Hello, World" application as communication.



interacts with the outside world by running functions to do the input/output (Figure 3). During the I/O statements, the program is blocked. This is becoming a multisystem-spaced world in which the expectation is that the state can be observed by another program. The program and its languages should have notation and concepts to share data dynamically at runtime, with no additional engineering.

A stand-alone application control has two elements: the forward and return movement of control, and the transfer of data during these movements. In the current programming model, because of the limitation of the state machine of the processor (with its current program counter), the forward and the return movements of control and data are synchronous (that is, the caller halts until the call control is handed back to the caller).

Now fast forward to the futuristic patient-doctor encounter mentioned at the beginning of this article. In this instance, the different devices, also known as agents, talk to each other to move the process forward. It is not a single-host system but a collection of distributed devices talking to each other, making a complete, whole self. In this new world, there are no statements or assignments—only a series of interactions, or sync points. How do you express this new world with no single point of control? How do you represent this new world with no obvious program control in the traditional sense?

To make this possible, let's explore the same stand-alone application and try to achieve the same control flow by delivering it as a network application. To network the example application, the three components—`caller`, `main`, and `print`—operate as three independent agents (Figure 4). Connecting these agents allows them to deliver the same print function in a networked setting. The caller agent kicks off the computation by sending data on the main action and waiting on `rtnMain` action. The `main` agent consists of two independent agents, one listening on the `main` action and the other listening on `rtnPrint`. The first component agent that receives the `main` action turns on **print**. Meanwhile, the one `print` agent listens on `print` and after printing on screen, turns on `rtnPrint`. The other main agent responds to `rtnPrint` and turns on `rtnMain`. Notice that each agent operates independently but is coordinated by the different actions or sync points. As **rtnMain** is triggered only after the print is completed, the functionality is the same as in the first C program. The difference is how the functionality is achieved through coordination of the autonmous agents: `CallerAgent`, `MainAgent`, and `PrintAgent`. These agents could also work across multiple system spaces.

Moving the application control from computation to communication makes applications work coherently across multiple system spaces.

Latency. As mentioned earlier, the simple C program that was a stand-alone application is now expressed as a network application by moving it from a premise of computation to one of communication. The sync points such as `main`, `rtnMain`, `print`, and `rtnPrint` coordinate these agents to create a coherent whole. These coordination elements could sit either in a single system space or across multiple system spaces. If these sync points sit across address spaces, then this introduces a new constraint: the latency of the network.

This now identifies the speed of the whole application. In a typical network application, latency is reduced when the application does not use the network. By introducing caching, the network usage is reduced, thereby increasing the speed of the overall application.

Scoping. Information hiding is an important property of a computer system. Programming languages should support this very well. To illustrate, object-oriented programming lets the programmer define the visibility of information to be either private or public. When private, information is visible only to the object. If the variable is declared as public, the information is visible to the whole program. These declarations are instructions to the compiler to control information visibility. The runtime information visibility has to be done by the programmer during design and construction. Having these kinds of information hiding in memory is fine for stand-alone applications, but with network applications, the language also should support scoping across the network. How is this accomplished?

In a network application, state transitions are exposed as sync points. In addition, information is transferred through the sync points. Clients can influence the application by interacting with these sync points. In Figure 5, the server has two sync points: `pa1`, which is hidden so it cannot be observed by the clients, and `pa2`, which is open and can be observed by all clients. In this configuration, only `client1` is aware of `pa1` and, hence, the server can be influenced only by `client1`. In terms of object-oriented principles, `pa1` is private and `pa2` is public. This principle is used to control information visibility across the network. Scope is controlled by hiding the action from the public and then giving the sync name selectively to different agents. Each of these clients and servers could exist across many address spaces, creating controlled information security over the network.

Network applications bring new challenges such as multisystem spaces, latency, intermittent network availability, and security. Thinking of applications as communication rather than as functions overcomes these challenges.

Mathematics of Communication Systems

Network applications should be developed with toolsets and languages that have communication as their foundation. Software based on computation has a good theoretical foundation, such as λ-calculus, on which the whole concept of programming is built. An application based on λ-calculus shuts out the world when the program executes, whereas a network application, by its very nature, has many agents interacting with each other on a network. This difference in the nature of the applications calls for a new foundation for the toolsets and languages for building network applications.

π-calculus,[6,7] the theory of communicating and mobile systems, is the algebraic representation of a Petri net. The focus in π-calculus is to define the communication between two agents that are operating on a network. It also can define how configurations of agents on the network change as they interact. This property of π-calculus—the ability to model interactions between many agents operating independently—makes it an ideal mathematical foundation for building toolsets and languages for network applications.

π-calculus achieves this by: (a) representing agent behavior as actions; (b) configuring the different agents as the initial starting configuration; (c) representing shared transition between these agents; and (d) changing the configuration of the agent network as state transitions happen. The π-calculus agent actions can be observed. An action represented by x is the potential to receive information. The action represented by $\bar{x}$ is the potential to be the source of information. The reaction of these complementary actions moves the process and the data from source to destination.

Figure 6 details initial configuration P. This is taken from page 88 of Milner's *Communicating and Mobile Systems: the Pi-Calculus*.[6] These terms run concurrently with each other. The parallel operation is identified by the operator "|". The sequence operation is represented by ".". The different actions are x and u. The action x carries through u. The action u transmits through v.

The π-calculus engine does interactions (that is, it looks at the action with the same name and executes the reaction). When the reaction happens, the value is communicated from one term to the other. In the case of the reaction between x and $\bar{x}$, it will send z through x. With the reaction, the second term becomes inert, represented by ∅—a term with no actions. Then, as shown in Figure 7, it proceeds to substitute u with z to raise the final configuration process state P1. Notice how the different term nodes collapse to create the final state.

It is possible to move programming from computation to communication—network-centric programming—by designing a language that is founded on π-calculus (such as, on interactions) and running it on a reaction engine that executes the language as reactions.

Figure 5. Scoping as communication.

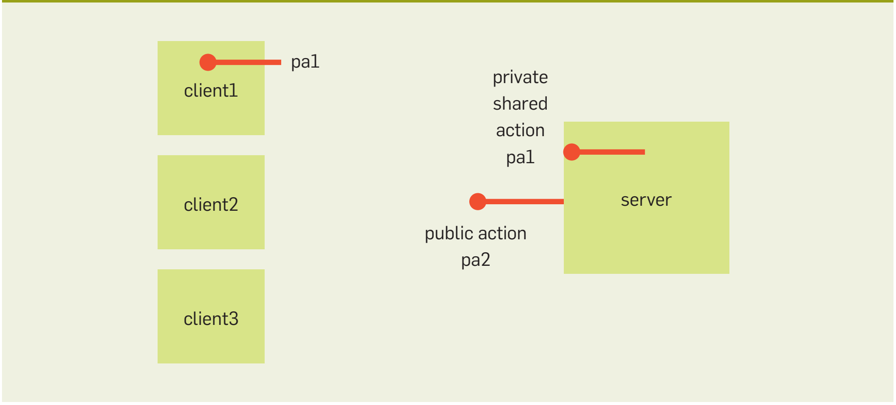


Figure 6. Initial processing configuration.

$$P = x(u).\bar{u} < v > \mid x < z >$$

Figure 7. Final process configuration.

$$P1 = \bar{z} < v > \mid 0$$

Figure 8. Interactions trace.

Current Interaction - Reaction between +ve and -ve	Active Ask Actions (-ve)	Active Tell Actions (+ve)
initial configuration initial configuration	`main, rtnPrint, print`	–
on interacting with `main`	`rtnPrint, print`	`print`
on interacting with `print`	`rtnPrint`	`rtnPrint`
on interacting with `rtnPrint`	–	`rtnMain`

Figure 9. Action definition.

```
<Actions>
  <Action>
   <Element>FirstName</Element>
   <Element>LastName</Element>
   <Element>EmailId</Element>
   <ActionName>View</ActionName>
  </Action>
  <Action>
   <Element>FirstName</Element>
   <Element>LastName</Element>
   <Element>MobileNumber</Element>
   <Element>EmailId</Element>
   <ActionName>Delete</ActionName>
  </Action>
  <Action>
   <Element>FirstName</Element>
   <Element>LastName</Element>
   <Element>MobileNumber</Element>
   <Element>EmailId</Element>
   <ActionName>Edit</ActionName>
  </Action>
  <Action>
   <Element>FirstName</Element>
   <Element>LastName</Element>
   <Element>MobileNumber</Element>
   <Element>EmailId</Element>
   <ActionName>Add</ActionName>
  </Action>
</Actions>
```

Hello, World Application as Interaction

Network applications are best addressed when based on communication. Unlike the functional model of computation (which has functions to move data from one variable to another) in a network application the movement of data is done by communication. As in physics, the flow of current is defined by the potential. Similarly, network applications are defined by setting up communication potential and are then wired together to create the spark that ensures the flow of information from the source to the sink.

The first step in building a network application is to define what flows when this spark happens. To define this action then defines which elements flow through it. Once the action is defined, then the two potentials for this action are set up by creating the +ve and -ve. The +ve is the source of information (like $\bar{x}$, in π-calculus), and the -ve (like x in π-calculus) is the sink of information. `Tell` sets up the +ve source of information, and `Ask` sets up the -ve sink of information.

The Hello, World network application has two agents that operate concurrently to deliver the functionality. `MainAgent` is composed of two sequences of two actions each. When this agent is activated, it activates `main` and `rtnPrint` simultaneously:

```
MainAgent = Compose [
      Sequence [ Ask(main);
      Tell(print)]; Sequence [
      Ask(rtnPrint);
Tell(rtnMain)]
      ]
```

The second agent is `PrintAgent`:

```
PrintAgent = Sequence [
Ask(print); Tell(rtnPrint) ]
```

The Hello, World application consists of these two agents operating concurrently. It is defined as

```
HelloWorldApplication =
Compose [ MainAgent;
PrintAgent]
```

Figure 8 shows the different actions that are on when each action is interacted with. To begin with, all agents become active and the actions `main`, `rtnPrint`, and `print` are exposed on the **Ask** side (-ve). The **main** action would be exposed, so it can be interacted with from a user interface. When `main` is interacted with, it activates `print` on the `Tell` side (+ve). The same name, `print`, is now active on both the +ve and -ve sides, setting up the situation for a spark. Next the interaction happens on `print` opening up `rtnPrint` on the `Tell` side (+ve). This leads to interaction with `rtnPrint`, leading to the next set of interactions. The interactions' trace is shown in Figure 8.

These actions are represented as REST URIs (uniform resource identifiers), so they can be interacted with over the network. Network programming is as simple as bringing agents into configurations with the objective of precipitating these interactions.

A Simple CRUD Application as Interaction

MasterKube software technology[5] is built with this network application in mind. The MasterKube software paradigm is used to build a simple CRUD application.

Define action. The first step is to define all actions that will be exposed to the outside system. These can be observed by any client UI that understands the MasterKube XML response. In this case there are actions to `Add`, `Edit`, `View`, and `Delete`, as illustrated in Figure 9.

Define behavior. Agents provide the dynamic part of the application. The agent `AddContactAgent` in the following code shows the action Add, which, when interacted with, creates a new agent. The `AddContactAgent` shows the `Add` UI. On interacting with the `Add` action, it creates the `Contact`, and `AddContactAgent` is repeated to show the same action (see Figure 10).

When there is interactions with `Edit`, it shows with all the elements that are in the `Edit` action. These are expected to be entered. On accepting these values into the system, `EditContactAgent` is repeated. Repeating the same agent gives the choice of the `Edit`, `Delete`, and `View` actions again, thereby exhibiting a persistent behavior.

In the `Delete` action the `Choice` action is empty. As there are no observations, interaction with `Delete` leaves no further observations and, hence, the contact is considered deleted.

All these actions are exposed as REST URLs, which can be interpreted by any device.

Figure 10. Agent definition.

```
<Agent>
<AgentName>AddContactAgent</AgentName>
  <Sequence>
  <!--Triggers the contact agent when add action is pressed -->
  <AgentCommand> <AgentName>ContactAgent</AgentName>
  <Ask><action><actionName>Add</actionName></action></Ask>
  </AgentCommand>
  <!-- Because this agent is in a sequence operation,
   the same agent is called recursively -->
  <AgentCommand> <AgentName>AddContactAgent</AgentName> </AgentCommand>
  </Sequence>
</Agent>
```

The `ContactAgent` accepts the `FirstName`, `LastName`, `MobileNumber`, and `EmailId`. It activates the `EditContactAgent`. The `AvatarName`, which is how the agent is identified, is the `LastName`.

```
<Agent>
<AgentName>ContactAgent</AgentName>
  <ProcessName>Contacts</ProcessName>
  <AvatarName>LastName</AvatarName>
  <Element>FirstName</Element> <Element>LastName</
Element><Element>MobileNumber</Element><Element>EmailId</Element>
  <Compose>
  <!-- Repeatedly show the edit contact agent till the user presses Delete
-->
  <AgentCommand><AgentName>EditContactAgent</AgentName></AgentCommand>
  </Compose>
</Agent>
```

The `EditContactAgent` shows the different actions shown by the contact. The different actions that are enabled for each contact are `Edit`, `Delete`, and `View`. These actions are shown in a choice, as only one action can be taken. Upon interacting with one option, the other actions go away.

```
<Agent>
<AgentName>EditContactAgent</AgentName>
  <Compose>
  <!--Two choices are presented. One is to Edit and the other is to De-
lete-->
   <Choice> <ChoiceOption>
     <Ask><action><actionName>Edit</actionName></action></Ask>
     <OptionAction>
     <Compose>
     <!--When edit is interacted with the same agent
      is repeated to show persistence -->
     <AgentCommand> <AgentName>EditContactAgent</AgentName> </AgentCommand>
     </Compose> </OptionAction>
   </ChoiceOption>
   <ChoiceOption>
     <Ask><action><actionName>Delete</actionName></action></Ask>
     <OptionAction> <!--When delete is interacted with no further
      observations are exposed so this contact disappears. --> </OptionAc-
tion>
   </ChoiceOption>
   <ChoiceOption>
     <Tell><action><actionName>View</actionName></action></Tell>
     <OptionAction>
      <!--When View is interacted with
        the same agent is repeated to show persistence -->
     <AgentCommand><AgentName>EditContactAgent</AgentName></AgentCommand></
OptionAction>
   <//ChoiceOption></Choice>
  </Compose>
</Agent>
```

Conclusion

The miniaturization of devices and the prolific interconnectedness of these devices over high-speed wireless networks are completely changing how commerce is conducted. These changes (a.k.a. digital) will profoundly change how enterprises operate. Software is at the heart of this digital world, but the software toolsets and languages were conceived for the host-based era. The issues that already plague software practice (such as high defects, poor software productivity, information vulnerability, and poor software project success rates) will be more profound with such an approach. It is time for software to be made simpler, secure, and reliable.

Moving software from its computing foundation to a communication foundation will ensure the promise and benefits of digital are more widely shared. C

Related articles on queue.acm.org

A Guided Tour through Datacenter Networking
Dennis Abts and Bob Felderman
http://queue.acm.org/detail.cfm?id=2208919

You Don't Know Jack about Network Performance
Kevin Fall and Steve McCanne
http://queue.acm.org/detail.cfm?id=1066069

The Network's NEW Role
Taf Anthias and Krishna Sankar
http://queue.acm.org/detail.cfm?id=1142069

References

1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641; http://dl.acm.org/citation.cfm?doid=359576.359579.
2. Fielding, R.T. Architectural styles and the design of network-based software architectures. Ph.D. dissertation, 2000, University of California, Irvine; https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.
3. Futures and promises. The Boost C++ Libraries; http://theboostcpplibraries.com/boost.thread-futures-and-promises.
4. Kernighan, B., Ritchie, D.M. *The C Programming Language*. Prentice Hall, 1978.
5. MasterKube Software Development Manual. 2014
6. Milner, R. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
7. Milner, R. Elements of Interaction. *Commun. ACM 36* 1 (Jan. 1993), 78–89; http://dl.acm.org/citation.cfm?id=1283948.
8. ReactiveX; http://reactivex.io/intro.html.
9. Tanenbaum, A.S., van Renesse, R. Distributed operating systems. *ACM Computing Surveys 17*, 4 (1985), 419–470.

Antony Alappatt (antony.alappatt@masterkube.com) is the founder of MasterKube, whose goal is to provide path-breaking software technology and services to serve the new digital era.