Category Overview

8/12/07

Coordination

One out of many

How do ducks maintain a “V” formation when flying? How do bees organize their hives and their search for pollen? How do humans and machines jointly cooperate to operate eBay, the online auction company? How do customers and web site robots interact to fill a virtual shopping cart and complete transactions? How do you and I agree on a request and fulfill a promise?

These are all coordination questions. In each, many entities interact within constraints to produce a mutual outcome that no one could produce alone. How does this happen?

Coordination is as fundamental in computation as it is in other parts of nature. We build multi-threaded programs to control multiple processors working together to solve a problem; the multiple threads are an opportunity to speed up the computation by a considerable amount. We routinely use computers to coordinate large projects and businesses in the Internet. We build computations in the Internet that interact with humans and other computations to carry out business; eBay, for example, is an automated system that manages a large number of auctions and their associated sales. We built vast multi-player games, such as Second Life with 8.6 million residents, that not only entertain but produce useful results.

The challenge in computing is to understand coordination algorithmically. Only then can we successfully simulate natural coordination systems. Only then can we successfully delegate human tasks to machines. Even when we think we have automated coordination properly, small mistakes can mess it up completely. In February 2007, a fleet of new F-22 fighter jets all lost their navigation systems because of a programming error as they crossed the international date line on a test flight; fortunately they were able to visually follow a leader back to base in Hawaii.

Not surprisingly, the computational understanding of coordination has fed back into everyday life, yielding improved business transactions, greater trust, new games, and more productivity.

We make a distinction between coordination and communication. Coordination refers to cooperative action; communication to the transfer of messages. While distinct, they are not independent. The sign of successful communication is often a successful coordination among the parties.

A Model for Coordination Systems

Perhaps the most powerful model for coordination systems is the "game". The players make up a "society" where players follow rules and strategies and use resources in ways that contribute to the common objective. The players can be humans, computational agents, or both. The elements of a game are:

(1) **Players:** The agents (including people) who interact in the system. There can be one (as in solitaire) or many. There can be rules of eligibility: only certain types of people or levels of skill are allowed in the game.

(2) **Objectives:** The goals that the players are collectively trying to achieve. There are two types of game, finite and infinite (James Carse, Ballantine Books, 1986). The objective of a finite game is for someone to win (and thus terminate the game). The objective of an infinite game is to continue the play. For example, a ball game or video game is finite; the players seek to have someone declared the winner by scoring the most points. The market economy is an infinite game; the players come and go, and their objective is to keep the economy going and growing. The Internet is likewise an infinite game.

(3) **Resources:** The various tools and items of equipment that the players need for their moves. For example, the resources of chess are a board, black pieces, and white pieces. The resources of a baseball game are the playing field, the bases, the balls, and the bats. The resources of the market economy are communication systems including Internet, banks, transportation systems, and offices.

(4) **Rules:** A rule is a statement that authorizes or limits actions. An authorizing rule can be expressed C:M, where C is a condition and M is a move (action); the move is allowed whenever the condition holds. A limiting rule can be expressed M:C; the move is allowed except when C is true. In the game of driving, "turn right on red after stop" is authorizing, and "do not cut someone off" is limiting.

(5) **Strategies:** A strategy is a guideline for series of actions that are known by experience to help attain the objectives of the game. In chess, exchanging pieces of similar point values is a good strategy. In driving, watching the brake lights of cars two or three lengths ahead is a good strategy.

This model calls attention to an important aspect of coordination: all actions are initiated by individuals, and the common objective is realized only when individuals collectively coordinate in the right way. The rules help the individuals achieve coordination, while still having latitude to make their own choices.

It is possible to constrain the game so much by having no-choice rules that the game then behaves like a programmed system: there is only one outcome. The games of tic-tac-toe and nim are like this. Games are much more interesting when players have many choices leading to many different outcomes. Then the strategies become useful to guide them toward the goal despite the uncertainties.

Understood in this way games are a deep concept. They are not simply means of entertainment, they go straight to the heart of coordination.

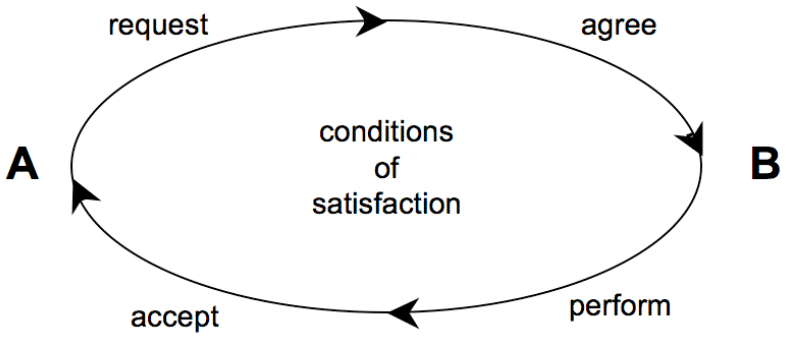
A closed computation is like a finite game, an open computation like an infinite game. In its early days, computing was concerned primarily with closed computation (finite games). As the Internet has matured, computing is now concerned primarily with open computations (infinite games).

In computing, we are interested in games where (1) some or all of the players are computational agents, and (2) interactions are mediated by computations. Let us consider first what it means to delegate human tasks to computational systems. Then we will consider two complementary ways to approach mediation issues.

Action Loops

The most fundamental human coordination pattern is the action loop. This is a conversational protocol in which one person fulfills a promise to the satisfaction of another. The backbone of the protocol between the parties Alice (A) and Bob (B) is:

A: I request.
B: I accept.
B: I deliver.
A: I am satisfied.



Alice's request includes a statement, the proposed *condition of satisfaction* (CoS), of what is to be provided. After a negotiation, Bob agrees to a (possibly modified) CoS. After a period of performance, Bob declares that the CoS are fulfilled. Alice reviews Bob's work and declares satisfaction. Much human coordination depends on the success of the parties at completing loops of this form, over and over again.

Action loops are pervasive in natural systems. Human and animal neural systems contain many local loops in which a signal moves in one direction and a confirmation in the reverse. The coordination dances of ants and bees are built of action loops. Animal social systems exhibit hierarchies in which more dominant individuals consolidate their power by initiating more action loops.

Action loops arise naturally in the design of computers. It is very common to create a request-acknowledge loop between two hardware components: the one sends a request signal to the other, which responds with an acknowledge signal. This scheme prevents the initiating component from sending another request until the other component is ready. Some email systems track action loops by observing which speech acts are transmitted in the exchange of messages.

Levels of Delegation

Many computations automate tasks that humans formerly did. When we design a computation to carry out a human task, we say that the human task is delegated to the computation. A human coordination task can be delegated by automating part or all of the action loops involved.

Here's how the delegation of a single loop goes. Define the state of the loop to represent the amount of progress toward completion. The

loop starts out in State 0; it moves to State 1 when Alice makes the request, to State 2 when Bob accepts, to State 3 when Bob delivers, and returns to State 0 when Alice declares satisfaction. The four speech acts of the loop trigger the state changes. The entire loop can be delegated to a simple state machine.

Because human coordination systems are composed of many interacting action loops, it may now appear that every human coordination system can be completely delegated to a computation.

But this delegation is necessarily incomplete. There are "contextual aspects" of human interaction that cannot be easily represented, and therefore delegated. For example, humans experience various emotions and moods; some of these (for example, ambition and urgency) help move the parties toward completion, while others (for example, anger and distrust) hinder. There is no way to account for the effects of emotions and moods in a state machine.

Therefore, delegation of tasks to computations is not necessarily easy and requires a considerable understanding of how humans coordinate and what they assume tacitly.

Delegations of human tasks to computations occur at three basic levels:

(1) **HH: human to human.** Alice and Bob interact directly. Their interactions are tracked by a computation, whose records help them monitor each task's progress toward completion. The field of Computer Supported Cooperative Work (CSCW) has grown up to address these types of interactions. The humans do the work, while computations track and record completed and remaining work for them.

> For example, Action Technologies created a system that allowed companies to draw a map of all the roles, interactions, and action loops involved in a business process. The map was compiled into a database representation. As individual transactions moved through the network, individual role players could see the requests arriving at their (virtual) desks, where they could process them and dispatch them to the next station in the network. This system handled all the routing and tracking of tasks, while leaving the decisions and judgments to the human actors.

(2) **HC: human to computer.** Bob (performer) is delegated to a computation. Alice interacts with the computation through an

interface. The field of Human Computer Interaction (HCI) has grown up to address these types of interactions.

For example, NASA has long been interested in flight simulators. They built systems that present the pilot with scenes that might appear through the airplane's windows. Moving the controls causes the scene to move in exactly the way it would appear as the airplane responds. Advanced simulators include motion generators that move the pilot in a chair to simulate horizontal and vertical motions of the aircraft. The objective is that the pilot learns to respond to situations in the right ways, prior to actually encountering them in real flight. NASA has extensively investigated how to design simple displays that convey complete situational information to the pilot; how much detail is needed in the scenes displayed or in the motions generated; and what actions the pilot cannot learn (e.g., the "feel" of a helicopter stick in stable flight).

(3) **CC: computer to computer.** Alice and Bob are both delegated to computations. The computations interact mainly with each other and occasionally with humans on the outside. The field of Concurrency Control (CC) has grown up to address these interactions. Concurrency control is critical in systems such as networks, operating systems, databases, robotic systems, and many more.

For example, operating systems provide automatic context switching and round robin scheduling to create the illusion that many computational processes (threads) exist in simultaneous execution. They provide semaphores so that one of these processes can signal another that it is safe to pass a checkpoint. They use these tools to create listener processes that monitor network ports for incoming packets and route them to the subsystems that process those packets. For example, an incoming file transfer packet is routed to the TCP subsystem for aggregation with other received packets making up a transmitted file.

Level 3 requires the greatest understanding of coordination, Level 1 the least. The deeper the delegation, the stronger the requirement for a computational understanding of coordination.

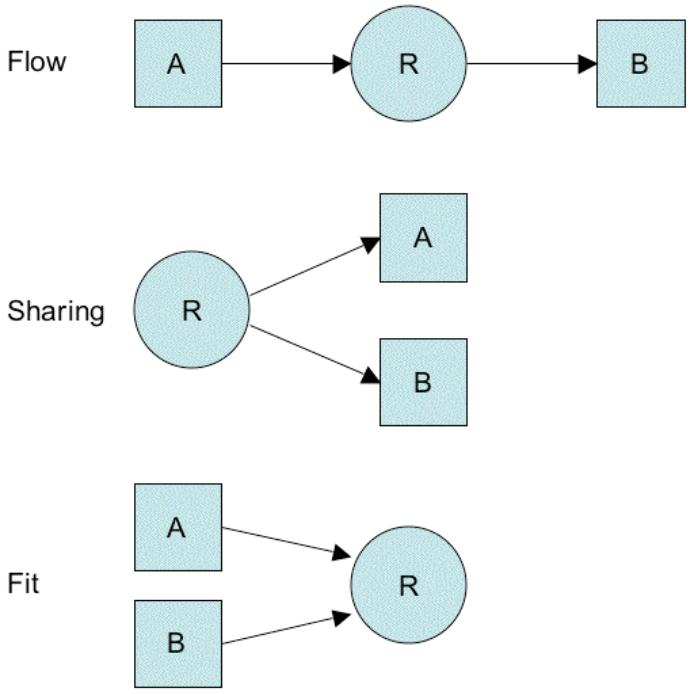
Dependency Patterns

The generic action loop captures the form but not the meaning of a two-person interaction. Most conditions of satisfaction are arranged around a small number of "dependencies". Tom Malone of MIT defines coordination as the management of dependencies among activities.

An activity is a set of tasks performed by a human or computational process. A dependency exists between activities A and B when the completion of one (say B) depends in some way on the other (A). For example: (1) Event A must precede event B. (2) B needs information from A before acting. (3) A and B both need to use the same processor. (4) A and B produce parts that are combined into a single assembly. (5) B adapts (customizes) to the profile of A. (6) A's input to B must be in formats recognized by B.

The figure below shows three possible dependencies for two activities and a resource. They are called flow, sharing, and fit.

- *Flow dependencies* arise whenever one activity produces a resource that is used by another activity. This common dependency includes message exchange, signaling, and flowcharting.
- *Sharing dependencies* occur whenever multiple activities all use the same (usually limited) resource. The future behavior of activities depends on the resource. For example, travel agents will offer seats only on published flights to their clients. If the resource is limited, the future behavior may also depend on what other activities are doing with the resource; for example, B may be forced to wait until A releases shared memory.
- *Fit dependencies* arise when multiple activities collectively produce, contribute to, or update a single resource. This kind of dependency arises when several engineers are designing different modules of a software system, when an assembly line is fitting parts into a car, or when different travel agents are booking seats on the same flight.



The flow dependency, which accounts for the majority of coordination mechanisms, has three subdependencies: (1) The *prerequisite dependency* concerns the timing of the flow -- how it is initiated (e.g., push or pull) or how often it is initiated (e.g., on a schedule or on demand). (2) The *accessibility dependency* concerns how the resource is made available to the activity that uses it (e.g., A ships it to B, or A makes it at B's location) (3) The *usability dependency* concerns making sure the resource is usable by B (e.g., the resource might meet some widely shared standard or A and B might negotiate the specifications individually each time). These three subdependencies correspond to the three elements of the common business phrase, "right thing in the right place at the right time"; in fact, they offer a rigorous working definition of this intuitive but often-imprecise business term.

This interpretation of coordination is especially useful for design. By identifying the essential activities and resources of a system, and drawing diagrams to represent their flow, sharing, and fit dependencies, the designer can then select the best means of implementing each dependency. The table below gives examples of coordination processes that implement the various dependencies.

Dependency	*Examples of coordination processes*
Flow	
Prerequisite ("right time")	• Make to order vs. make to inventory (pull vs. push) • Pre-defined schedule or ad hoc hierarchical control
Accessibility ("right place")	• Ship by various transportation modes • Assemble at point of use
Usability ("right thing")	• Conform to standards • Negotiate individual requirements • Participatory design
Sharing	• FIFO queueing • Preemptive priority queueing • Budget allocation • Managerial decision • Market-like bidding
Fit	• Predefined standards ("plan") • Case by case negotiation ("emerge") • Slotted synchronization • Mutual exclusion locks • Resolve conflicts by common manager or peer negotiation

These examples show that the coordination mechanisms we see in computational systems are the managers of dependencies; but they are not the actual dependencies. Dependencies are a higher level language and of implementations a lower level language. A designer first expresses the dependencies and then implements coordination mechanisms to manage them. The remainder of this overview examines the operational (implementation) level.

Essential Elements Of Coordination Systems

The game model applies at all three levels of delegation of human tasks to computations. No matter what else they do in the game, the players (humans or their agents) are constantly dealing with five fundamental coordination issues. We have developed computational methods for all five.

One-on-one interactions. Defined earlier, the action loop models the basic interaction between two agents who fulfill a mutual condition of satisfaction together. The game consists of many interdependent action loops whose individual conditions of satisfaction aggregate into the game's objectives.

The four components of an action loop rely on synchronization, a lower-level form of coordination. Synchronization means that one agent stops and waits until another agent signals it has passed a checkpoint. An action loop is defined by four synchronization events: the request, the promise, the delivery, and the acceptance.

Choice. Agents are constantly faced with multiple alternative moves, but they can only select one for action. This is problematic if the alternatives are equally attractive or if they are presented at (nearly) the same instant. The problem was characterized centuries ago by the philosopher Jean Buridan, who around 1330 in a commentary about free will, observed that a dog could starve to death if placed equidistant between equally attractive portions of food.

The choice-making problem is well recognized today by hardware designers, who have to deal with signals that can arrive unpredictably from many sources. It is possible for a standard flipflop (the basic storage element for a bit), on receiving simultaneous signals, to enter a metastable state from which there is no definite exit time. It is possible that the flipflop is still metastable at the next clock tick: the circuits that read it behave erratically because they cannot see a definite "0" or "1" from the flipflop. That in turn can cause the entire circuit to malfunction or lock up.

This was a real problem for computers well into the 1980s. The processor is supposed to be interrupted so that it can respond to external signals, for example, a new packet from the network, or a mouse click. But if two signals arrive simultaneously, the processor must choose one or the other and act on it first, returning to the deferred signal later. The flipflop that tells the CPU whether or not an interrupt signal has arrived may enter a metastable state if the signal coincides with the clock. If that flipflop is still metastable at the next clock tick, the CPU will not read a definite "0" or "1" and may hang, requiring rebooting. The aggregate probability of such failures can be quite high. In a 24-hour period, a computer with clock speed 10^9 ticks per second will exercise the decision circuit about 10^{14} times. If the design criterion is no more than one decision crash per day, the architect needs a circuit with failure rate less than 10^{-14}. It's hard to make hardware that reliable!

The same problem occurs with any agent that is asked to make a definite choice among alternatives within a deadline. The agent's brain, or the social system with which the agent interacts, may enter a metastable state whose exit time may be long. An example is the common situation of two people about to collide on a sidewalk. On sensing their imminent collision, they both stop. They exchange eye

signals, gestures, head bobs, sways, dances, and words until finally they reach an agreement that one person goes to the right and the other to the left. This could take a fraction or a second or minutes. They then resume walking and pass one another without a collision. The key is that the two parties stop for as long as is needed until they decide.

The impossibility of guaranteeing a choice within a deadline is called the Choice Uncertainty Principle. Most computational mechanisms have to deal with this principle, because at some level they are required to choose between simultaneous events.

It is sometimes alluring to believe that software solves the problem that hardware cannot. In software you can express that, when two processes want simultaneous access to a serially reusable resource, you make them set a lock before access. Your proof that the locks choose only one process at a time to enter the critical section implicitly assumes that only one CPU at a time can gain access to the memory location holding the lock value. If that's not so, then occasionally your critical section will fail no matter how careful your proofs. Every level of abstraction at which we prove freedom from synchronization errors always relies on a lower level at which choice arbitration is solved. But choice arbitration can never be solved absolutely.

Feedback. Agents rely on various feedback mechanisms to inform them of the effectiveness of past actions. A common feedback mechanism is the simple message exchange, such as in an action loop. The performer (Bob) gets direct feedback from the requester (Alice) and can assess whether the delivered result is satisfactory. If it is not, Bob can retry. If Bob is repeatedly unsatisfactory, Alice can find a different performer. Another common feedback is a performance metric, which is a numerical quantity derived from statistical analysis of data collected from sensors in the system. For example, a store owner discontinues items that don't sell well, and increases inventory of items that do.

Racing. When two or more agents attempt to read or write the same resource at the same time, the values they see (or record) may depend on the relative order of their access, and can therefore be unpredictable if the order is unpredictable. To see how this might cause a problem, consider a husband and wife accessing their bank account simultaneously from two ATMs. Husband wants to deposit $100 and wife wants to withdraw $200. Shortly after they initiate their transactions, both ATMs will have uploaded a copy of the account record. Husband adds $100 to his balance and wife subtracts $200 from hers. Then, depending on which ATM writes its copy of the

record back to the bank last, the bank will show either a change of +$100 or -$200, but not the proper change of -$100. The only way to avoid this is to make sure that one transaction runs to completion before the other can start. That requirement is called serialization. Serialization means that if two agents try to access the same resource together, they are forced to go in order, one at a time. Notice that serialization assumes that the selection problem has been solved: if the requests are simultaneous, we want to select just one and defer the other. Serialization also means that when it is selected, each agent gets exclusive use of the resource.

Many computing systems include instructions that implement serialization through a "mutual exclusion lock". An agent that needs exclusive access to a resource for a limited period obtains the lock before accessing the resource, and releases the lock when done. Other agents test the lock and wait if they find it locked. Again, note that the lock instruction must solve the choice selection problem so that two simultaneous agents do not erroneously both think they have set the lock.

In some situations, serialization is not enough. The outcome of a system of properly serialized agents still depends on their relative order. (An example is airline seat assignment: whoever requests a filled seat cannot get it.) There are many large multi-agent computations in science, engineering, or commerce, that use parallel processes for high performance. These computations must give the same result no matter what the relative speeds of parallel agents. This property of a system is called determinacy: the final outcome depends only on the initial inputs to the system.

To achieve determinacy, we add to serialization a second principle, called the overlap principle. The overlap principle says that no two concurrent agents may access and modify resources together. The overlap principle eliminates any possibility of racing, either a race in which they both write a value in the same resource, or a race in which one reads a value previously written by the other. A system of agents obeying the overlap principle will be determinate.

Deadlocks. In many multi-agent systems it is possible under the right circumstances for a set of agents to become entangled in a circular wait. That means each one is stopped waiting for another in the set to send a signal. Since everyone is waiting, no one can actually send a signal. Deadlocks are moderately difficult to detect and can be even harder to avoid.

On example of deadlock arises when two agents request two resources in different orders; thus A asks for (R1,R2) and B for (R2,R1). If they

both initiate their requests at the same time, they can enter a state where A holds R1 and B holds R2. When A then asks for R2, it must wait since B already holds R2. When B asks for R1, it must wait since A already holds R1. Now both A and B are waiting for the other to release a resource. They are both stuck. Neither can do anything. The only way out is to abort them and start over.

This sort of deadlock can be avoided by requiring that all agents request resources in some preset order. In the previous example, both A and B would be required to request in the order (R1,R2); they cannot both wind up waiting for the other to release a resource.

Requesting out of order is not the only way to generate a deadlock. Two agents can get into a deadlock simply because of an insufficiency of resources they have been authorized to use. Suppose the bank gives A a credit limit of 10,000 and B a limit of 15,000; but the bank only has 20,000 in its loan pool. A can ask for 8,000 (OK because it's less than A's credit limit) and then B can ask for 12,000 (OK because it's less than B's credit limit). However, the loan pool is now exhausted. If A returns to ask for another 2,000 it will have to wait. If B returns to ask for another 2,000 it will also have to wait. Now there is a deadlock because neither A nor B can release any funds back to the loan pool. The only way out is to foreclose one or more of them, releasing their funds back to the pool, until requests of the remaining waiting agents can be satisfied.

These examples easily generalize to any number of agents and any number of resources.

The best strategy for deadlock avoidance is to establish operating rules that prevent the circular wait. One possibility, already noted, is to require all agents to request resources in the same order. Another possibility is to make sure that the common pool of resources is large enough to accommodate all possible future requests; this was not the case in the bank loan example. A third possibility is to grant all agents all the resources they need at the very start; they never have to stop and wait again. None of these possibilities is universal. They work in some cases but not others. Therefore, many systems contain deadlock detectors, invoked occasionally to determine whether a stopped agent is part of a circular wait.

Livelock is a related issue that can arise when the computational processes are too accommodating: everyone winds up deferring to the others. This pattern has been called an after-you-after-you wait. An example is record locks in a database. A transaction requests locks one at a time and releases them all if it encounters a locked lock; lock contention over a popular record can make all transactions loop

indefinitely. A common solution is to use “exponential backoff” -- on finding a lock locked, the transaction initially waits a random time before retrying; on each retry, if it is still locked, it doubles the retry wait.

Implementations

The protocols and methods for dealing with these generic coordination problems are sufficient for a wide range of agent systems.

Some systems require multiple parallel processes even though the underlying hardware has a single CPU. This is common in multi-tasking operating systems, which set up many separate processes to manage independent activities. A typical operating system, for example, has background processes that listen for events in the TCP, FTP, and HTTP protocols. Many Java programs contain multiple threads that can be doing many things simultaneously. These operating systems contain a scheduling module that uses a combination of context switching and time-slicing to create the illusion that many processes are executing concurrently. The scheduling module multiplexes multiple CPUs among many processes, accommodating multi-core CPU chips. It should be no surprise that the scheduling module has to be designed to react properly to simultaneous events and not be thrown into a lockup due to a metastable state.

Even with such basic tools for process coordination, it is still possible to make severe programming errors. For this reason, programming language designers have provided higher-level abstractions such as atomic transactions, lockable objects, monitors, and communicating sequential processes. The compiler translates the high level program into a lower level program that includes the proper coordination operations.