Message Passing for Distributed QoS-Security Tradeoffs

Hala Mostafa[1], Partha Pal[1] and Patrick Hurley[2]

[1]*Raytheon BBN Technologies, Cambridge, MA 02138*
[2]*Air Force Research Laboratory, Rome, NY 13441*
Email: hmostafa@bbn.com

Information Assurance (IA) is a growing concern, since almost every aspect of our lives depends on distributed information systems and the frequency and sophistication of cyber attacks targeting these systems is on the rise. However, IA cannot be considered in isolation, as it also affects the Quality of Service (QoS); with limited resources, the security mechanisms employed for IA (e.g., firewalls, antivirus, encryption) usually adversely affect QoS levels delivered by a system. The system therefore needs to make tradeoffs between IA and QoS. These tradeoffs are complicated by the facts that users' relative preferences over QoS/IA change based on the situation, the preferences of different users conflict, and tradeoff decisions made at one node in the distributed system typically affect other nodes as well. We address the problem of distributed computation of tradeoffs among various aspects of QoS and IA in a way that maximizes the satisfaction of all stakeholders.[a] Specifically, we want the nodes in the system to make coordinated decisions as to what local actions to take to optimize QoS/IA levels delivered by the system. Our first contribution is formulating this problem as a Distributed Constraint Optimization Problem (DCOP). This entails quantifying various notions involved in tradeoffs to be able to compare options in the course of optimization, as well as encoding the effects of various decisions on the quantities we want to optimize. The DCOPs we obtain have cost functions where multiple local configurations result in the same cost. In addition, the corresponding factor graphs contain many cycles. To deal with these issues, our second contribution is a value propagation algorithm that helps nodes reach a consistent set of decisions even in cyclic factor graphs with non-unique local optima. We present experimental results comparing the performance of the max-sum algorithm with and without value propagation against other algorithms when applied to domain-inspired and random instances. On the domain-inspired instances, max-sum with value propagation achieves near optimal solutions at a fraction of time taken by DPOP, with the reduction in solution cost due to value propagation most pronounced in scenarios with resource contention. On random instances, value propagation obtains solutions with cost 1.7x of optimal, even when performed without a utility propagation algorithm first.

[a]A shorter version of this work appeared in the Optimization in Multi-Agent Systems Workshop, 2012 [1].

Approved for Public Release; Distribution Unlimited: 88ABW-2012-4884, 10-Sep-2012

Keywords: Distributed constraint optimization, information assurance, value propagation

1. INTRODUCTION

An increasing number of security mechanisms are being used to defend modern distributed information systems (e.g., firewalls, antivirus scanners, access control, encryption) against malicious cyber attacks. *Information Assurance (IA)* is concerned with ensuring that the security mechanisms are effective, and the system can be entrusted with critical information processing tasks. Traditionally, IA has taken an all-or-nothing approach where the entire system is deemed secure or insecure, mostly depending on the user's perception. This approach results in lost opportunity when a system is declared insecure and all activities are halted, but in reality only parts of the system have been compromised and some activities could have progressed without security breaches. This approach also pushes the users to take undue risk, because there are times when the user must act, and even though their actions (e.g. installing a software or disabling a security mechanism) may diminish the overall security

of the system, the impact is not reflected in the security state perceived by other users.

More recently, runtime assessment of IA has been advocated, where variation of the system's level of assurance due to attack-induced failures, environmental threats (e.g., release of a new virus), and user-made changes are taken into consideration [2]. Runtime assessment also reflects the fact that security requirements themselves are changing and depend on where the system is within the mission its users are trying to accomplish.

Security mechanisms use the same system resources (e.g., CPU, memory, network bandwidth) that are needed by the information system they aim to defend. As a result, IA and Quality of Service (QoS) interfere, often adversely. For example, increasing the strength of encryption consumes more CPU resources and increases the round trip response time for service request and response, thereby negatively affecting the availability and timeliness (QoS attributes) delivered by the system. In mission-critical applications of distributed information systems (e.g., network centric warfare, telemedecine, internet voice/video applications), degraded QoS may mean loss of service, with impacts ranging from loss of revenue to loss of life.

This contention over resources necessitates a tradeoff between the IA and QoS levels delivered by a system. There are three complicating factors in this tradeoff. The first is that there are typically multiple users (stakeholders) of the system. These stakeholders can have conflicting requirements and preferences. Left to themselves, the competing and conflicting requirements of different stakeholders, even when they are participating in the same mission, can result in degraded performance where less important requirements are met at the cost of more important ones. The second factor is that an individual user's relative preferences for QoS and IA are not static, but depend on where he is within the mission he is trying to accomplish. For example, an intelligence analyst may prefer to have high definition video, but if there is an external threat, he can make do with standard definition if it is over an encrypted channel. There will also be situations where one aspect of QoS (or IA) is preferred over another. For example, a black and white video with low frame drop rate may be preferable to a color video with dropped frames. The third complicating factor is that tradeoff decisions made at one node in the distributed system can affect QoS and IA levels at other nodes. For example, the decision to use a low bandwidth encrypted channel may affect the bandwidth available to other services communicating over this channel.

In the QIAAMU project [2, 3], we are developing algorithms and supporting infrastructure to enable user-specific requirement-based runtime management of QoS and IA in a unified way. Our *Continuous Mission-oriented Assessment* approach relies on existing security and system management infrastructure to collect measurements to assess whether the delivered levels of QoS and IA meet the user's requirements. Complementing the assessment, runtime management of QoS and IA also needs to take or suggest remedial actions when available resources cannot meet all QoS and IA requirements of all users.

We address the problem of distributed tradeoffs in the context of QoS-IA contention over resources. More specifically, we calculate tradeoffs between various aspects of QoS and IA in a way that maximizes the satisfaction of all stakeholders given limited resources. For each decision-making node in the distributed system, the decision problem we consider is: *what local actions should each node take to optimize QoS/IA levels delivered by the system as a whole.*

We make two main contributions in this paper. First, we formulate the decision-making problem of the nodes in the distributed system as a Distributed Constraint Optimization Problem (DCOP). To do this, we begin by quantifying various notions involved in tradeoffs so that the utilities associated with different decisions can be compared in the course of optimization. We also need to encode the effects of the various decisions on the quantities we want to optimize. In our case, the effects are more complex than the constraint functions generally used in DCOP literature. We propose a representation, *cause-effect network*, which is a deterministic Bayesian network that captures relations between the actions taken by the nodes and the various QoS and IA attributes they affect.

Our second contribution is a message exchange algorithm to be used after utility propagation (e.g. using the max-sum algorithm) whose goal is to handle the special characteristics of the DCOPs resulting from QoS and IA tradeoff scenarios. Our DCOPs have the following characteristics: 1) the corresponding factor graphs contain many cycles of various lengths, 2) some nodes have large degrees, 3) multiple configurations of variables minimize any given function, potentially resulting in locally optimal, but globally inconsistent choices. Our proposed value propagation algorithm helps nodes reach a consistent set of decisions in spite of the cyclic nature of the graph and the non-uniqueness of decisions that optimize the QoS/IA levels at any one node.

The rest of the paper is organized as follows: Section 2 gives a background on QIAAMU, our distributed systems runtime assessment and management framework. Section 3 briefly reviews DCOPs and the max-sum algorithm. Section 4 introduces cause-effect networks and details how we formulated the distributed QoS-IA tradeoff problem as a DCOP. We present our value propagation algorithm in Section 5. Experimental results comparing multiple DCOP algorithms to our value propagation algorithm (combined with max-sum or stand-alone) are given in Section 6. Finally, we conclude and highlight directions for future work.

2. THE QIAAMU FRAMEWORK

The level of Information Assurance (IA) delivered by a system has traditionally been estimated using offline analyses, penetration testing, modeling and experimentation. At run-time, arguably the time when it is most critical to be assured about the system, IA takes an all-or-nothing approach and is mostly dependent on the user's perception (i.e., either the user continues to believe the offline assessment or not). In a distributed system, the network is a shared resource whose load and failures are mostly unpredictable, and assumptions made offline about the operating condition of the system may often be invalid in the deployment environment. Consequently, there is a need for continuous re-assessment of IA levels delivered by a system. A similar argument holds for Quality of Service (QoS) levels. In the QIAAMU project [2], we are developing algorithms and supporting infrastructure to enable user-specific requirement-based runtime assessment and management of QoS and IA. We have described our metrics and assessment framework in previous papers [2, 3] . In this paper, we focus on the decision making involved in the runtime management of QoS and IA.

Figure 1 shows fragments of an example distributed system that we will use to explain the components of the QIAAMU framework. There are 2 MRAPs (Mine Resistant Ambush Protected vehicles) sweeping the route between a FOB (Forward Operating Base) and the COP (Combat Outpost). The MRAPs communicate with headquarters (HQ) through a satellite link (SAT) or through the FOB that communicates with HQ over the Internet. The former is more secure, but the latter has higher bandwidth. The MRAPs, FOB and COP employ a number of security mechanisms.

The components of the QIAAMU framework are:

- **Nodes:** These are the computing and communication entities in the distributed system. A subset of these nodes (the ones in Figure 1) run the QIAAMU framework and are therefore able to coordinate with others on decisions to improve QoS-IA levels. Each node running QIAMMU is responsible for managing the IA and QoS requirements of one or more stakeholders using actuators are under its local control.
- **Actuators:** Actuators are used to effect changes in the system. An actuator can pertain to the use or configuration of some security mechanism (e.g. setting a firewall policy or using an antivirus), or some QoS decision (e.g. communicate over a high bandwidth channel rather than SAT). The sets of actuators managed by different nodes are non-overlapping, but an actuator on one node can affect QoS/IA levels delivered at another node.
- **Stakeholders:** The stakeholders are cooperating users of the system, supporting a common mission objective in different roles. In our example, the stakeholders are the war fighters on each MRAP, the commander on HQ and a system administrator. Each stakeholder has an associated importance factor that reflects the importance of his/her tasks to the achievement of the overall mission, and therefore determines how hard the system will try to accommodate this stakeholder's requirements and preferences.
- **Attributes:** An attribute is an aspect of system performance that is of interest to some stakeholder. Some attributes pertain only to IA (e.g., Confidentiality and Integrity), some only to QoS (e.g., Timeliness and Fidelity), and some to both QoS and IA (e.g., Availability).
- **Requirements:** A stakeholder can specify a requirement for a particular attribute of a particular asset. An asset can be a database, a service or a communication channel. As we detail later, requirements are expressed in terms of *levels*. For example, a user can require that the Confidentiality of the HQ database be high and its Availability be medium. Because resources are bounded and not every attribute will attain its highest level, the system gives the stakeholders a means of expressing their satisfaction with less-than-perfect levels (e.g., by stating that the requirement for an attribute is *Low*).
- **Preferences:** These can be used to specify a stakeholder's relative interests in different attributes of different assets. For example, the system administrator may care about the Integrity of the HQ database more than the Timeliness of the instant messaging service, while another stakeholder may have a preference for the latter.
- **System/environment conditions:** these are quantities that we can measure about the system or environment that are relevant to determining how well a system is doing in terms of QoS/IA. Examples include throughput on a communication channel, number of unauthorized login attempts and whether a given host is reachable. It is the combination of system/environment conditions and configuration of actuators that determines the level of QoS/IA attributes delivered by a system. For example, when there is a virus threat (an environment condition) and the antivirus policy actuator is set to strict, the level of Integrity can still be regarded as high. Stated differently, system/environment conditions at the time of application affect the results of applying an actuator; if there is no virus threat, there may be no benefit in stricter antivirus policy in terms of whether a stakeholder's IA-related requirements are met. Similarly, if a node has been up for a long time (an environment condition), rebooting the node may have a positive effect on its health. Because the desirability of an actuator

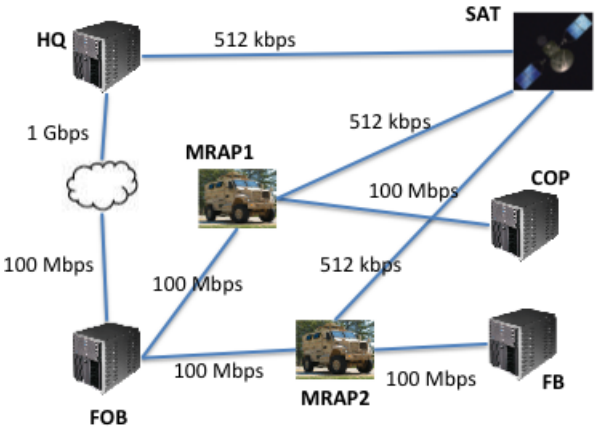

FIGURE 1. The MRAP scenario

configuration depends on conditions at the time of actuation, any decisions about actuators must be made in the context of current conditions.

The specific problem we address in the overall runtime QoS and IA management framework is this: given a set of system/environment conditions, how should each node configure its actuators such that the satisfaction of all stakeholders across all nodes is maximized, weighted by their relative importance?

The QIAAMU framework uses the list of components detailed above in the following ways:

- **Assessment:** In this process, each node populates its system/environment conditions by taking actual measurements of the system. For example, the "CPU load" system condition can be populated with the output from the `top` command, while the "number of unauthorized login attempts" can be populated by parsing a log file produced by a connection-monitoring process. The delivered levels of the various attributes are then determined from the combination of actuator configurations and system/environment conditions. These levels are then compared to stakeholder requirements to determine which ones are met.
- **Tradeoff:** The tradeoff refers to the decision making process by which the distributed system as a whole trades off QoS and IA levels in order to maximize the satisfaction of the stakeholders. The result of a tradeoff is a configuration for each actuator in the distributed system. Consider a very simple example with two nodes, where one node controls the length of the encryption key used by both nodes, and the other controls the channel over which they communicate. In a setting where Availability is more important than Confidentiality, the tradeoff can result in the first node choosing a short key (to decrease CPU load and positively affect Availability) and the second node choosing the channel with the higher bandwidth and lower inherent security. Notice how a decision by one node can also affect other nodes in the system.
- **Actuation:** In a distributed system that is fully under the control of our QIAAMU framework, a tradeoff would be followed by actuation where the solution actuator configuration is actually enacted (e.g. by re-configuring firewall policies). However, because many operational systems are not fully automated, the result of the tradeoff can be displayed to a human operator as a recommended course of action.

3. BACKGROUND: DCOP AND MAX-SUM

In a Distributed Constraint Optimization Problem (DCOP), we have a set of n variables $\mathbf{x} = \{x_1, x_2, .., x_n\}$ and a corresponding set of finite domains $\{D_1, D_2, ..., D_n\}$. Each of m constraint functions (constraints) involves a subset of variables (its *scope*) and maps each configuration of these variables to a cost. A *local* assignment for function f_i is an assignment of values to variables in its scope. A *global* assignment is an assignment to all variables in $\mathbf{x}$. We formulate our problem as a minimization problem where the goal is to find a global assignment that minimizes the global objective function, usually a summation of the constraint functions. Formally, the goal is to find a global assignment $\mathbf{x}^*$ such that

$$\mathbf{x}^* = \arg\min_x \sum_{i=1}^{m} f_i(\mathbf{x_i})$$

where each f_i is a constraint function and $\mathbf{x_i}$ are the values of variables in its scope. There may be more than one assignment that minimizes the global objective function.

3.1. The max-sum algorithm

The max-sum algorithm is a message-passing algorithm that has been widely used to solve DCOPs [4]. It is a member of a family of algorithms relying on the Generalized Distributive Law. Max-sum operates on a factor graph representation of the DCOP where there is a function node for each constraint function and a variable node for each variable. There is an edge between a variable x_i and a function f_j if the former is in the scope of the latter. Nodes in the factor graph exchange two kinds of messages:

- Message from a variable to a function

$$q_{i\to j}(x_i) = \alpha_{i,j} + \sum_{k\in\mathcal{N}_i\setminus j} r_{k\to i}(x_i)$$

where $\mathcal{N}_i$ is the set of functions that are neighbors of variable x_i in the factor graph, and $\alpha_{i,j}$ is a normalizing constant that prevents the values in messages from increasing indefinitely in a cyclic factor graph.

- Message from a function to a variable

$$r_{j\to i}(x_i) = \min_{\mathbf{x_j}\setminus i}\left[f_j(x_j) + \sum_{k\in\mathcal{N}_j\setminus i} q_{k\to j}(x_k)\right]$$

where $\mathcal{N}_j$ is the set of variables that are neighbors of function f_j in the factor graph.

If the factor graph is a tree, max-sum is guaranteed to converge to the globally optimal value assignment. Moreover, it can find this assignment in only two sweeps of the tree. The first sweep proceeds from the leaves of the tree upward, with each node getting messages of one of the types listed above and sending messages of the other type. In the second sweep, each variable node x_i locally determines its optimal value by calculating the following function from the messages it received:

$$z_i(x_i) = \sum_{j\in\mathcal{N}_i} r_{j\to i}(x_i) \quad (1)$$

and setting itself to the value that minimizes this function.

One of the attractions of max-sum is that the small message size which scales with the size of the domains, as opposed to algorithms where message size is exponential in the number of variables. However, max-sum is not guaranteed to converge in cyclic graphs, and when it does, the solution it finds is not guaranteed to be optimal. Nevertheless, the literature gives pointers to work where empirical results show that it performs well on cyclic graphs (e.g., pointers in [4, 5]).

The max-sum algorithm is closely related to the belief propagation (BP) inference algorithm for graphical models. They compute messages in the same way to achieve related goals; max-sum operates on cost functions represented by the constraints to find an assignment that minimizes global cost, and BP operates on local probability distributions to find an assignment that maximizes joint probability; the Maximum A Posteriori (MAP) assignment. The motivation behind using max-sum in graphs with cycles comes from practical success in using loopy BP (LBP) on graphical models with cycles. There have been theoretical results on the performance of LBP on graphs with various topologies. For graphs with single loops, it was shown that LBP converges to a stable fixed point or a periodic oscillation. If it converges to a fixed point, the fixed point messages give rise to the MAP. For graphs with arbitrary topologies, Weiss and Freeman [18] show that a fixed point of LBP gives rise to an assignment that is the MAP within a certain neighborhood they define. However, they state that they assume a unique global MAP assignment. As we detail later in the paper, our QoS-IA tradeoffs give rise to DCOPs with multiple local and global optima.

3.2. Other DCOP algorithms

As we show in Section 4, our DCOPs have factor graphs that contain cycles, and their constraint functions have non-unique minimizers (the minimum of Eq (1) is attained at more than one value). The basic max-sum algorithm is not guaranteed to reach optimal solutions under these conditions, so the values chosen according to Eq 1 can give rise to an inconsistent global solution. In the following paragraphs, we discuss variants of max-sum and other DCOP algorithms that are concerned with these issues.

The issue of cyclic graphs is addressed by Distributed Pseudotree Optimization Procedure (DPOP) [6], which is a follow up for an older algorithm, DTREE [7], that is only correct for trees. In fact, DTREE on trees performs the same operations and exchanges the same messages as max-sum. In a tree, the utility reported by a sender node to its parent only depends on the subtree rooted at that node, which is not the case in a cyclic graph. DPOP creates a pseudotree from the graph where a node's utility can depend on one or more variables *above* the parent that are connected to the sender by *back edges*. The UTIL messages exchanged in DTREE are therefore not adequate in pseudotrees, since they only report the optimal utility for the subtree for each value of the parent. DPOP's UTIL messages report utility for each value combination of *all* parents of a node; the parent through a tree path as well as parents on back edges. Non-leaf nodes then proceed to combine and send UTIL messages up the tree. Value propagation is largely similar to that of DTREE.
DPOP is optimal, even on general graphs. Unfortunately, although the number of messages exchanged by DPOP is linear in the size of the tree, the size of a UTIL message can be exponential in the induced width of the pseudotree. There have been several variants of DPOP that try to address the message size issue [8].

Another approach to dealing with cycles is to remove them. Bounded max-sum [9] is an algorithm that removes cycles by eliminating dependencies between functions and variables which have the least impact on solution quality and subsequently uses max-sum on the resulting spanning tree. The result is a bounded approximation of the optimal solution of the original problem. However, when constraints are not binary (involve more than two variables), the algorithm's choice of which dependencies to eliminate may not be the one that minimizes the impact on solution quality. As we state in Section 4, the constraint functions in our DCOPs have a large arity, so it is not clear that this approach will work well in our domain.

Our value propagation algorithm is somewhat similar to value propagation in Max-Sum-AD [10]. In that work, the cycles in the factor graph are addressed by assuming an order over the nodes and allowing messages to flow in only one direction according to the order. After convergence in one direction, the algorithm

proceeds in the opposite direction. Breaking ties and dealing with inconsistent assignments is addressed using a subsequent value propagation phase. Experimental results are only reported for binary constraints.

The work on Multi-Objective DCOPs (MO-DCOPs) [11] also deals with cyclic graphs with non-unique minimizers. In MO-DCOPs, non-uniqueness is due to the presence of multiple objective functions, which can give rise to multiple Pareto optimal assignments that do not dominate each other. The cycles are addressed by bounded max-sum. The non-uniqueness is addressed by a value propagation phase executed on the cycle-free factor graph computed by the bounding approach.

In summary, cyclic graphs can be dealt with by having a sophisticated utility propagation phase and a relatively simple value propagation phase (e.g., DPOP), or having a simple utility propagation phase but a sophisticated pre-processing of the graph into a tree (e.g., bounded max-sum). As we detail in Section 5, our approach is to do neither pre-processing nor sophisticated utility propagation, but to have a sophisticated value propagation phase. We believe that the advantage of this approach is that our value propagation phase can be used with any utility propagation phase (not necessarily max-sum). However if, as in the case of DPOP for example, the utility propagation phase is sophisticated, we suspect that our value propagation will be of questionable benefit. We see the largest impact of our value propagation phase either as a stand-alone coordination algorithm (as will be demonstrated in Section 6) or following a utility propagation phase that results in variables converging to sets of local optima, in which case it allows them to adopt values that are part of the same global optimal.

4. FORMULATING THE DISTRIBUTED TRADEOFF PROBLEM

In this section, we detail the steps of obtaining a DCOP formulation for the problem of IA-QoS tradeoff in a distributed system. The solution to the DCOP specifies how each node should configure the actuators under its control such that the satisfaction of all stakeholders across all nodes is minimized, weighted by their relative importance.

4.1. Quantization and quantification

To be able to compare the desirability of different sets of QoS/IA attribute levels, we need to express how important each attribute is to each stakeholder, and how important each stakeholder is to the overall mission of the system. We therefore associate with every stakeholder $\boldsymbol{s}$ in the set of stakeholders $\mathcal{S}$ a weight $\boldsymbol{w_s}$ indicating the relative importance of this stakeholder. Similarly, we express the relative preference of stakeholder s to a particular attribute of a particular asset as $\boldsymbol{p_{s,a}}$ where a refers to the attribute of an asset (e.g. Availability of database1).

TABLE 1. Parameters of the QoS-IA tradeoff DCOP

x	A set of values to all actuators
e	A set of values to all environment/system conditions
s	A stakeholder
a	An attribute of an asset
w_s	Weight (importance) of stakeholder s
$p_{s,a}$	Preference of s to attribute a
$q_{s,a}$	Requirement of s for the level of attribute a
$v_x^e(a)$	Level of attribute a from actuators setting x and environment/system conditions e

Note that values of these parameters are typically going to be specific to a mission mode (a time interval within a mission). We therefore assume there is an implicit subscript M that indicates which set of values is currently being used. In the rest of the paper, we address decision making for a given mission mode, i.e., under a given set of parameters.

We also express attribute levels (both delivered and required) in terms of quantized and ordered levels. This quantization facilitates knowledge elicitation from the stakeholders and its representation in our formulation. The requirement level desired by stakeholder s for attribute a is denoted by $\boldsymbol{q_{s,a}}$.

Because environment/system conditions modulate the effects of actuators on attributes, we denote the level of attribute a delivered by setting the actuators as dictated in the value assignment $\boldsymbol{x}$ under environment/system conditions $\boldsymbol{e}$ by $\boldsymbol{v_x^e(a)}$. Table 1 summarizes the quantities and symbols in our formulation.

4.2. Cause-effect networks

To be able to compare the desirability of different actuator configurations, we need to elicit from the domain expert how QoS/IA attributes depend on the settings of actuators and the current environment/system conditions, i.e., we need to elicit the various $v_x^e(a)$ functions. However, when each attribute depends on a large number of actuators in a complex way, it becomes tedious and/or difficult for a user to specify each function as a flat table detailing how, for example, each configuration of firewall policies, choice of communication channel and several other factors affect the Availability of HQ. It may be much easier to specify how Availability depends on Throughput and Reachability and how these depend on the actuators and system conditions.

We therefore use a representation of the set of functions $v_x^e(a)$ that is similar to Bayesian networks but with deterministic, rather than probabilistic, relations. Our *cause-effect networks* are directed graphs with nodes representing variables and edges representing cause-effect relations. Each non-root node has a table, which we call the Conditional Value Table (CVT), a

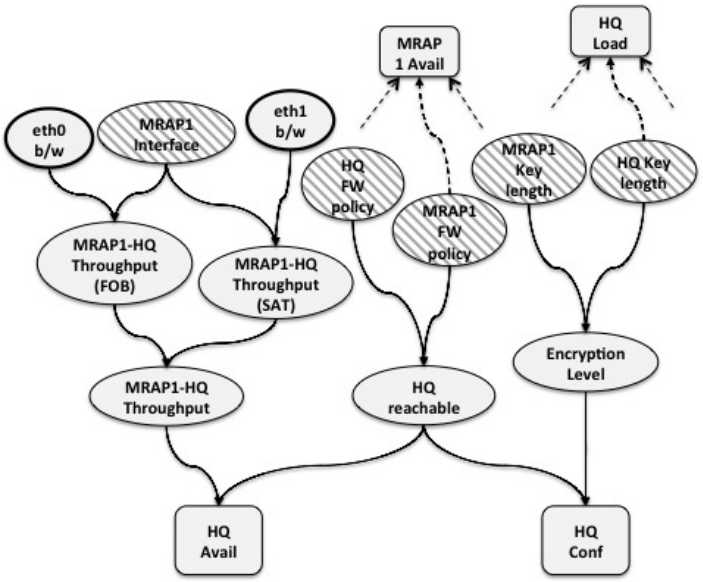

FIGURE 2. Part of the MRAP cause-effect network. Hashed nodes represent actuators, ovals with thick borders represent system conditions, clear ovals represent intermediate values and boxes represent attributes. Dashed arrows represent other influences affecting an attribute.

deterministic version of conditional probability tables, that specifies how its value depends on the values of its parents.

Figure 2 shows part of the cause-effect network for the MRAP scenario. It shows how decisions made by MRAP1 and HQ ultimately affect the Availability and Confidentiality of HQ. The decision-making nodes in this example are MRAP1 and HQ. Nodes with dark borders represent environment/system conditions (e.g. the current bandwidth of the links on eth0 and eth1 interfaces). Hashed root nodes represent actuators; the encryption key length and firewall policies used by MRAP1 and HQ, the interface MRAP1 uses (eth0 to FOB or eth1 to SAT). Intermediate nodes represent intermediate calculated values (e.g. throughput of the MRAP1-HQ connection). Finally, leaves represent attributes (e.g., Confidentiality of HQ). When actuator settings and environment conditions are plugged in at the roots of the network, values of nodes at subsequent levels can be calculated from their respective CVTs, ultimately determining the values of the leaves that represent the QoS/IA levels delivered by the system under the environment conditions and chosen actuator settings.

This representation has the advantages of making explicit the structure of causes and effects among the variables. Not only is it efficient in terms of space, it is also helpful in terms of knowledge elicitation. Directly specifying how each actuator configuration affects the attributes of interest is analogous to writing out the joint probability table rather than representing it using a Bayesian Network.

4.3. DCOP formulation

We formulate the problem of finding the configuration of actuators that maximizes the satisfaction of all stakeholders as a DCOP. [3] The variables are the actuators in the system, and each variable's domain is the set of values the corresponding actuator can take. For example, the domain of the "Encryption key length" actuator can be the set {128,256,512,1024}.

The goal is to find a global assignment that minimizes the difference between the required and delivered levels of attributes for all stakeholders weighted appropriately. The objective function is the penalty incurred when these two levels do not match and can be formulated as:

$$\min_{x} \quad \sum_{s \in \mathcal{S}} w_s \sum_{a \in \mathcal{A}} [\, p_{s,a} * \max(0, q_{s,a} - v_x^e(a))]$$

where the symbols are as in Table 1. The *max* operator expresses the fact that there is no benefit to having the delivered level exceed the required level; the best case is when these two levels match and the penalty is 0.

The above objective function can be factored into a sum over cost functions (the DCOP constraints), each of which is associated with an attribute-asset pair (e.g. the Availability of the HQ database). The cost function of pair a is therefore

$$f_a(x) = \sum_{s \in \mathcal{S}} w_s * p_{s,a} * \max(0, q_{s,a} - v_x^e(a))$$

Figure 3 shows the factor graph for the partial cause-effect network in Figure 2. As we mentioned earlier, the graphs resulting from QoS-IA tradeoff problems have many cycles of various length and some function nodes with a large number of neighbors. In addition, the CVTs in the cause-effect network oftentimes give rise to constraint functions with non-unqiue minimizers. For example, HQ Availability has the same value for multiple assignments of MRAP1's interface and firewall policy and HQ's firewall policy.

As the e in the above formula suggests, the cost function is calculated in the context of a given set of environment/system conditions. Different sets of conditions give DCOPs that are structurally the same, but differ in their cost functions, since they differ in the functions $v_x^e(a)$.

As can be seen, our cost functions are more complex than typical functions (e.g., in graph coloring). A cost function of a given attribute-asset pair is actually a cause-effect network where values of the root nodes (actuators) are plugged in and propagate through the network to yield a value for the attribute-asset pair. In the course of solving the DCOP, each function

[3] Our problem, and DCOPs in general, have some similarities to Collaborative Design Networks (CDNs) [12], but CDNs are probabilistic. Solution approaches proposed for CDNs bear strong similarity to max-sum and assume agents are arranged in a hypertree.

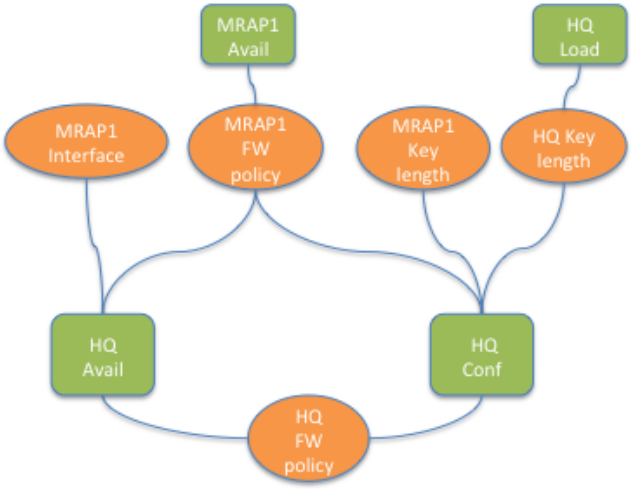

FIGURE 3. Part of the MRAP factor graph. Ovals represent variables (actuators) and rectangles represent functions (attributes).

will typically be minimized many times under different assignments of the variables in its scope. To reduce the cost of the large number of such minimizations, we avoid re-evaluating a function from scratch; we only propagate values from the root variables that changed since the last evaluation and only re-evaluate the descendants of these roots.

5. VALUE PROPAGATION

Initially, we used the max-sum algorithm as described by Farinelli et. al [4] to solve our DCOP. Our choice was motivated by initial attempts at using DPOP, which proved too slow and often failed to solve our instances because it ran out of memory (more details in Section 6). As discussed earlier, the fact that a node in our factor graph typically has a large number of neighbors exacerbates DPOP's exponential message size.

When we experimented with max-sum, we found that the combination of a cyclic graph and non-unique locally optimal assignments resulted in the agents reaching local assignments that sometimes have low global quality. However, it is important to note that this low quality is not due to max-sum failing to converge; in our experiments, utilities converged within a maximum of 15 iterations. The reason for the low quality solutions we sometimes obtained is the widespread presence of ties among utility values. This results in variables picking local assignments that are part of different optimal global assignments and are therefore inconsistent with each other.

We experimented with adding small random noise to the values in the utility messages to break ties among the non-unique locally optimal partial solutions. We observed that even though ties were broken, they were broken by each node locally, which still didn't address the problem of inconsistent assignments which resulted in solutions with poor global quality.

An acyclic graph with non-unique local minimizers can be handled by a simple value propagation phase that proceeds from the node designated as the root variable to the leaves (as suggested in [5]). After calculating its optimal value, the root node passes this value down. A function that gets a value from its parent calculates (or retrieves a stored copy of) the corresponding optimal values of its children and propagates them. A variable that gets a value from its parent passes it on to its children.

The above procedure cannot be directly applied to graphs containing cycles, since there is no longer a notion of parents and children. Even if we enforce a topological order on the nodes, a variable can potentially have multiple parents, in which case it is not clear what value it should take. Other approaches overcome this problem by constructing a DFS tree for the cyclic graph and performing value propagation on the tree.

To overcome the above problems, we propose a value propagation phase which can be used either on its own or following a utility propagation phase (e.g. max-sum) that narrows down domains of variables. Our proposed scheme operates on the factor graph representation of a DCOP. Functions optimize in the context of the (possibly reduced) variable domains and suggest values to their neighbors. Because the graph is cyclic, a variable can receive multiple suggestions, so it decides which suggestion to adopt using a heuristic measure of function importance. If a variable ignores a function's suggestion, the latter re-minimizes in order to obtain consistent values for the other variables in its scope.

We start by explaining the notion of function importance, followed by details of our value propagation algorithm. We then compare it to DSA [13] as an example where agents coordinate by exchanging values, and end the section with a workout of a detailed example.

5.1. Function importance

During the course of our value propagation algorithm, we will need a measure of function importance to break ties and help variables decide which value suggestion to accept. Our initial heuristic was to determine importance based on the number of neighbors a function has (its degree), with the rationale that it is easier for a function with a smaller scope to optimize in the context of values suggested by a function with a larger scope, rather than the other way around. However, we realized that regardless of scope size, it is a function's contribution to the overall solution cost that matters. We therefore calculate the weight of function f_a related to attribute a as follows

$$w(f_a) = \sum_{s \in \mathcal{S}} w_s * p_{s,a} \quad (2)$$

which reflects the importance of the stakeholders who care about attribute a and how much they care about it. In our experiments, we only report results using this heuristic, since it performed consistently better than the degree-based one.

5.2. Value propagation

As mentioned earlier, our value propagation phase can be used as a stand-alone coordination algorithm, or following a utility propagation phase that results in variables having many values minimizing the function z_i in Eq 1. In either case, if each variable just goes ahead and chooses one of these values, the result may be a globally inconsistent solution. The goal of a value propagation phase is to assign values in a consistent manner.

The value propagation (VP) phase starts as follows (if utility propagation is not used, we skip steps 1 and 2 and proceed directly with step 3):

1. Each variable narrows down its domain. It performs the minimization in Eq. 1 and retains only the minimizing value(s). If there is a single minimizing value, it is sent to neighboring functions in a VP message. If there are multiple values (but not the entire domain), we call this set of values *candidates*. If all the values in a variable's domain have the same utility, then this variable is indifferent among its values and therefore does not take part in initiating the VP phase.
2. If no variable has a single minimizing value, each variable that has candidates sends VP messages to its neighbors with *null* in the sender field to signify that this variable is not suggesting any values, but just wants neighboring functions to do their minimizations using the variable's reduced domain.
3. If no variable has candidates, the function with the highest importance is selected and sent an empty VP message.

To know if there are variables with single minimizers or candidate sets, the variables exchange their reduced domains.

In the following, we detail how each kind of node processes VP messages.

5.2.1. How a function processes VP messages

A simple way of processing VP messages is for a function f_i to calculate an assignment of the variables in its scope $\mathbf{x_i^*}$ that minimizes its cost, with the minimization taking place over the potentially reduced, rather than the entire, domains of variables (after step 1 above, each variable can end up with a domain containing a single value, a set of candidates, or the original set of values). The function would then send out VP messages to its neighbors telling them to take on the values in $\mathbf{x_i^*}$.

The problem with the above scheme is that in the presence of cycles, a variable may receive multiple conflicting assignments from its neighbors. We therefore consider a VP message from a function to a variable to be a *suggestion* rather than an enforcement and allow a variable to *decline* a suggested value if it has previously adopted a different value suggested by a more important function. A variable declines a suggested value by sending a VP message back to the sender containing the other value it has taken on.

Algorithm 1 shows how a function processes a batch of VP messages. The basic idea is that a function receiving VP messages re-optimizes in the context of the current domains of its neighbors. After calculating a new minimizing assignment, the function suggests the minimizing value to each neighbor not in `senders`. If a variable was merely declaring that it has candidates, it will not be in `senders` (because the *sender* field in its message was *null*, per Step 2 above) and so will receive the suggestion. A neighbor that did send a message does not need to receive a suggestion because the fact that it did so means that it declined a previous value; i.e., it already reduced its domain to one value, which is what the function used in the minimization.

Performance optimization: Because a function can receive multiple batches of VP messages, it can end up calculating $\mathbf{x_i^*}$ multiple times. However, some of these minimizations may be unnecessary if the messages will not change the function's opinion of what the values of its neighbors should be. We avoid unnecessary computations by having each function keep track of the optimal assignment (`curOptimal`) from the most recent minimization. The minimization only needs to be redone if 1) the sender variable is telling the function it has candidates or 2) the value in an incoming VP message differs from values in `curOptimal`, in which case the sender variable is telling the function that it has declined the function's previous suggestion.

Algorithm 1 Function.processVP(in: *msgs*)

```
1: for all msg ∈ msgs do
2:    if msg.sender != null then
3:       senders.add(msg.sender)
4:    end if
5:    storedDomains[msg.sender] = msg.value
6: end for
7: curOptimal = getMinAssignment(storedDomains)
8: for all neighbor do
9:    if neighbor ∉ senders then
10:      neighbor.processVP(curOptimal[neighbor])
11:   end if
12: end for
```

Another aspect of how functions process VP messages is the storing of the domains of certain neighboring variables, which we will explain after detailing the operation of variable nodes.

5.2.2. How a variable processes VP messages
Because a variable that adopts a value may later need to decline it, each variable keeps a list, `funsWhoAssigned`, of the functions that assigned it to its current value so that it can notify them if it later declines. Algorithm 2 shows how a variable processes VP messages. It starts by determining `topMsg`, the message whose sender function has the highest importance. If this sender is more important than any function in `funsWhoAssigned`, the variable adopts the value in `topMsg`, and if it is different from the current one, the list is cleared. The variable then sends VP messages containing its current value to a neighbor if a) it did not receive a message from this neighbor in this round (thus propagating the value to it) or b) the neighbor sent a message that conflicts with the new value (thus declining the value suggested in neighbor's previous message). The list `funsWhoAssigned` is re-built in the process, in case the variable needs to decline in the future.

Algorithm 2 Variable.processVP(in: *msgs*)

```
1: topMsg = msg with most important sender
2: topFun = topMsg.sender
3: if topFun.importance ≥ maxImportance(funsWhoAssigned) then
4:   if currentValue ≠ topMsg.value then
5:     funsWhoAssigned.clear()
6:   end if
7:   currentValue = topMsg.value
8: end if
9: for all neighbor do
10:  if neighbor ∈ senders and
     neighbor.msg.value = currentValue then
11:    funsWhoAssigned.add(neighbor)
12:  else
13:    neighbor.processVP(currentValue)
14:  end if
15: end for
```

5.2.3. Storing domains
When we implemented the above algorithms, we encountered situations where a function f that previously assigned a value to one of its neighbors v and has to re-minimize ends up doing so in context of the single value it assigned to v. Ideally, f should be able to 'undo' its assignment and consider v's entire domain when it re-minimizes. This can be achieved if each function stores domains of its neighbors and uses the stored domain if it wants to undo a value it previously assigned.

As an example of why we need to store domains, consider the situation shown in Figure 4 where f_1 is connected to v_1 and v_2, each of which is connected to other functions as well. Assume each variable narrowed down its domain to the candidate set $\{a, b, c\}$. Suppose

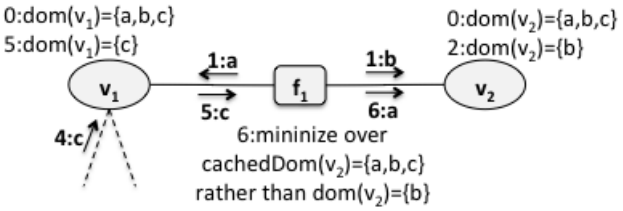

FIGURE 4. Importance of storing domains. Arrows show VP messages labeled with Time:Value

that at time 1, f_1 sent value a to v_1 and b to v_2. On receiving these messages, v_1 and v_2 set their domains to the values in their respective messages. At a later time, v_1 declined (because it received a suggestion, c, from a more important function) and sent back a VP message to f_1 with its new value. If f_1 re-minimizes, it will do so with the domain of v_2 restricted to b. Instead, what f_1 should use is the original set of candidates of v_2, if available, that potentially has a wider set of choices rather than just b. The key point here is that v_2 reduced its domain as a result of a message from f_1, so f_1 should be able to "undo" this reduction. f_1's stored version of v_2's domain is $\{a, b, c\}$, which indeed allows it to explore all these options in re-minimizing in the context of $v_1 = c$.

It is important to note that if v_2 had taken on a value suggested by a more important function than f_1, v_2 would have sent a VP messages to f_1 and in processing it, f_1 would have refreshed its store to reflect v_2's new domain and re-minimized in this new context.

A neighbor's entry in a function's `storedDomains` is updated when the neighbor sends a VP message to this function. Also, in `getMinAssignment`, if the function encounters a variable for which it does not have a stored domain, it queries the variable for its domain, uses it for minimization, and updates `storedDomains`.

So overall, storing its neighbor's domains results in the desirable behavior whereby f_1's re-minimization is done in the context of values chosen by more important functions, if they exist, or a broader set of options, if they do not.

5.2.4. Cycle detection
In terms of when messages are processed, we implemented a schedule where at the beginning of each round, an agent processes all the messages that were sent to it in the previous round in a batch. This is in contrast to an agent processing each incoming message individually and sending a set of outgoing messages in response.

In our implementation of max-sum, a node in the factor graph ignores a message from a sender if it is the same as the last message from this sender. In addition, each variable node keeps a history of states. A state is the set of most recent messages, one from each neighbor, and is updated after each batch of messages

is processed. A node uses its history to detect cycles; situations where it is re-visiting the same set of states, in which case the node ignores incoming messages and therefore produces no outgoing messages. The algorithm converges if no nodes have outgoing messages.

We terminate the utility propagation phase upon convergence or reaching a pre-set number of iterations. We found that cycle detection can slightly reduce runtime and the number of iterations. Predictably, it has no effect on solution quality. We used cycle detection for all the results we report for our implementation.

5.3. Comparison to DSA

The kind of value propagation we propose is somewhat similar to the Distributed Stochastic Algorithm (DSA [13]). In DSA, an agent uses the latest values received from its neighbors to evaluate the constraints it is involved in and if there is a value that improves its local state, the agent switches to that value with a certain probability p. If it changes its value, an agent sends the new value to its neighbors.

We believe the key difference between our VP and algorithms like DSA is in the fact that in our VP, functions, rather than variables, are in charge of assigning/suggesting new values. To see why this is important, consider a function f_1 involving 5 variables v_1 to v_5. In DSA, each of these 5 variables *independently* makes its evaluations in an attempt to find a value that reduces the cost of the function. In the presence of multiple minimizing assignments, if more than one variable changes its value simultaneously, the result may be an inconsistent global assignment. And if v_1, for example, changes its value to improve a function f_2 that it has with some other variable v_6, each of v_2 through v_5 will be responding to this change *independently*.

The above is contrast to our proposed approach where f_1 and f_2 are responsible for re-optimizing and suggesting values to variables. The advantage is that f_1, for example, has a much larger context for optimization than any individual variable node. As such, it will not suggest incompatible values, and if a variable declines a suggestion (as v_1 did), the function is in a better position to adjust the rest of the variables.

In Section 6, our experimental results clearly demonstrate the advantage that our VP has over DSA.

5.4. Value propagation example

We now present a simple example that highlights the value propagation algorithms followed by function and variable nodes. Consider a factor graph with 5 variables and 4 functions as shown in Figure 5. For simplicity, all variables are binary. Assume that at the end of the utility propagation phase, the utilities accumulated by variables v_3 and v_4 resulted in each having a single value that minimizes Eq (1) for it (this value is `True` for v_3 and `False` for v_4). Assume that because of non-unique minimizers, the other 3 variables cannot narrow down their domains and do not have *candidates*. Assume also that according to the weights calculated by Eq (2), $w(f_1) > w(f_2)$.

The iterations in the value propagation phase of our example proceed in the following steps.

- **Step 1:** because they have single minimizers, v_3 and v_4 send these values to their neighbors.
- **Step 2:**
  - f_1 gets the message from v_3, sets the stored domain for v_3 to $\{T\}$ and calls `getMinAssignment` to calculate the optimal complimentary values for the other variables in its scope. Assume this call returns $\{v_1 = T, v_2 = F, v_3 = T\}$. These values are stored in `curOptimal` and sent out to v_1 and v_2.
  - f_2 does a similar process where it optimizes in the context of the received message. Assume its minimizing assignment is $\{v_1 = T, v_2 = T, v_4 = F, v_5 = F\}$. f_2 sends out to v_1, v_2 and v_5 their respective values. Note that f_1 and f_2 do not send out values to the variables they just received messages from.
- **Step 3:**
  - v_1 receives messages from both f_1 and f_2. Because $w(f_1) > w(f_2)$, `topFun` is set to f_1. Because this is the first VP message received by v_1, `maxImportance` returns 0 and v_1 adopts the value suggested by f_1 (Algorithm 2, line 6). Because the value sent by f_2 is the same as `currentValue` (the condition on line 8 is met), f_2 is added to the list of functions who assigned this value, but no message is sent to it.
  - v_2 processes its incoming messages much like v_1 did, except that the value suggested by f_2 is not the same as that suggested by the top function f_1 (the condition on line 8 failed). v_2 therefore sends a message to f_2 containing the newly adopted value to let it know it declined f_2's suggestion.
  - v_5 receives a single message and adopts the value suggested in it. It does not send any messages.
- **Step 4:** f_2 receives v_2's decline message. Because the value in the message is different from that which f_2 calculated for v_2 in a previous round (the condition on line 6 of Algorithm 1 is met), f_2 sets `changed` to True as a signal that it needs to re-optimize in light of this new message. f_2 also refreshes the stored domain of v_2 to $\{F\}$. The re-optimization is done in the context of the stored domains of v_1 and v_2. Assuming it results in the setting $\{v_1 = T, v_2 = F, v_4 = F, v_5 = T\}$, f_2 sends out messages to v_5. To avoid unnecessary communication, f_2 does not send messages to v_1

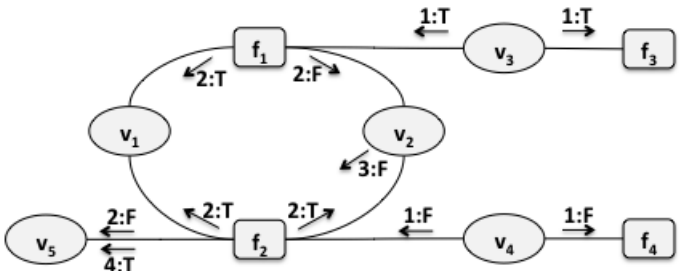

FIGURE 5. Value propagation example. Arrows show value messages labeled with Time:Value

and v_4, since neither of them got a new value as a result of the re-optimization, and they have not declined their previous values.

- **Step 5:** v_5 receives the single message from f_2. Since the previous message in Step 2 was also from f_2, the condition in Algorithm 2 line 3 is met. Because the value in the message is different from the current value, `funsWhoAssigned` is cleared. v_5 adopts the suggested value and no further messages are sent.

Let us now consider the same example without the value propagation phase. As with value propagation, v_3 and v_4 take on the values of their unique minimizers. But without value propagation, v_1 and v_2 pick values from their candidate sets independently of each other. They can potentially choose values that are inconsistent; i.e. when taken together minimize neither f_1 nor f_2. And even if they take on values that minimize the more important f_1, v_5 would take on the value F that minimizes f_2, without regard to the fact that F only minimizes f_2 if v_1 and v_2 are both set to T, which is not the case here.

To summarize, our value propagation phase has the following advantages

1. Handling conflicting value suggestions that arise in cyclic factor graphs by allowing a variable to accept/decline values suggested by neighboring functions based on a heuristic measure of function importance.
2. Allowing a function whose suggestions are declined to re-optimize in the context of values suggested by more important functions to produce values for other variables in its scope consistent with already assigned values.
3. Avoiding unnecessary restriction of domains and allowing a sort of backtracking through the use of stored domains in re-optimizing.

6. EXPERIMENTAL RESULTS

We conducted experiments to compare the following algorithms and variants:

- Max-sum without value propagation (*MS-NoVP*)
- Max-sum with value propagation (*MS-VP*)
- No max-sum, only the VP phase (*NoMS-VP*): We were seeing some cases where variables end up without any candidates after the utility propagation phase, so we wanted to test if there is value in doing max-sum utility propagation.
- Frodo's implementation of max-sum (*F-MS*), ADOPT [14] (*F-ADOPT*), DSA [13] (*F-DSA*), MGM [15] (*F-MGM*) and DPOP [6] (*F-DPOP*).

Frodo [16] is an open source framework for distributed constraint optimization that has implementations of several DCOP algorithms. We chose MGM and DSA as examples of simple algorithms that should work fairly well if the instances we test on are not too difficult. DPOP and ADOPT are dynamic programming and search-based algorithms, respectively, that guarantee optimal solutions and can thus provide us with benchmark solution quality.

For the settings of the Frodo algorithms, we set the number of cycles the algorithm runs for to 80 for F-MS, F-DSA and F-MGM. For F-DSA, the probability with which a variable changes its value is 0.5. The various heuristics used by F-ADOPT were left at their default values in the configuration file `ADOPTagent.xml` in the Frodo distribution.

For all algorithms, we report results of running on a single machine [4] where agents process their messages in sequence. For the timing results, we tried not to disadvantage the Frodo algorithms, so we solve all instances of a given DCOP family in the same Java Virtual Machine to amortize the initial time of loading the Frodo classes over the thousands of instances in each family. We used wall clock time (comparing time before and after the call to the `solve` method of an `AbstractDCOPsolver`) rather than simulated time. This being said, we realize that the engineering and implementation of Frodo is geared more towards extensibility than speed, so our timing results are more useful for comparing the Frodo algorithms against each other, and the impact of our value propagation phase on our max-sum implementation.

As for our algorithms, our implementation is entirely in Java. We run our implementation of max-sum utility

[4] 2.2GHz Intel Core i7 processor and 8GB of memory.

propagation until convergence or 80 iterations[5] have elapsed, whichever is earlier. But as mentioned earlier, except for very few instances (∼20 instances out of several thousands), our implementation of max-sum converged within a maximum of 15 iterations. For the settings where our VP is used, we run VP until convergence.

6.1. Domain-inspired scenarios

We hand-generated two sets of domain-inspired scenarios and their associated cause-effect networks. The first set consists of 6 scenarios based on the MRAP mission and the underlying distributed system. This set has relatively few actuators and attributes and not much contention over resources. The second set was developed with many more nodes in the factor graph and, as we detail below, more contention over limited resources by the various actors in the scenarios.

Each of these DCOPs represents the tradeoff problem the runtime management framework may face during a mission. Out of each cause-effect network, we generated a family of DCOPs by setting different values for the environment/system conditions and using different preferences of stakeholders over attributes. The DCOPs from a given scenario have the same factor graph, but differ in the constraint functions because the CVTs in the cause-effect networks are different (remember that environment/system conditions modulate the effects of actions). Each scenario resulted in thousands of DCOPs.

MRAP scenarios: This set of 6 scenarios (families of DCOPs) represents our initial efforts at modeling QoS-IA tradeoffs as DCOPs. They all started from the same set of core attributes and actuators, which we incrementally enlarged. The scenarios represent parts of the setting in Figure 1, but with a single MRAP that can reach headquarters through the FOB or the satellite. While each actor has its set of tradeoffs to make (e.g., FOB can use a strict firewall policy which increases its integrity but decreases availability), and different actors need to coordinate over common issues (e.g., MRAP and HQ need to use the same encryption key length), there are no shared resources over which different actors compete.

Figure 6 shows the factor graph of one of the MRAP DCOP families. Four decision makers (MRAP1, HQ, COP, FOB) decide on actuators like firewall policy (FW), antivirus policy, whether the system should reboot, communication protocol, encryption key length, process and user management (single/multiple user/process allowed at a time) and ethernet interface to be used. These actuators affect the integrity, availability, health and confidentiality of various assets in the distributed system.

[5]An iteration is a round where each agent receives the messages sent to it in the previous round and calculates its outgoing messages, which their recipients get in the next iteration.

The figure illustrates a recurring feature in DCOPs from QIAAMU scenarios, namely functions with large arities. For example, the function *HQ Avail & Conf*, which assesses the availability and confidentiality of headquarters, has 7 variables in its scope. We merge nodes that have the same set of neighbors. This has the advantage of reducing the number of cycles, without incurring the usual penalty associated with merging (an increase in a node's degree) because we only merge nodes that have the same neighbors. For variables, the result is a variable whose domain is the Cartesian product of its constituents, and for functions, the result is a function whose value is the sum of its constituents.

Table 2 shows the normalized times and solution costs produced by the various algorithms. To facilitate comparison across different DCOP families, we normalize the results with respect to the time and cost of the Frodo DPOP solutions; times are percentages of the time taken by F-DPOP and costs are percentages over the optimal DPOP cost (e.g., in Scenario 1, F-MS produced a solution whose cost is 13% higher than optimal in 70% of the time taken by F-DPOP).

As expected, our implementation of max-sum, although not optimized for speed, is faster than Frodo's implementation, due to the overhead in the latter. As for the quality obtained by our max-sum implementation (without VP) compared to the Frodo implementation, it is not clear to us why our implementation gives lower cost solutions. In the larger scenarios (scenarios 1, 2 and 6), ADOPT runs out of memory with 500M of memory and fails to terminate within a reasonable time and has to be aborted when we run it with 1.5G of memory. On the instance that ADOPT can solve, it takes much longer than DPOP. In most cases, local algorithms (DSA and MGM) took longer than complete algorithms and produced solution qualities comparable to our NoMS-VP setting.

Larger scenarios: This set of instances represents a more realistic setting with more attributes and actuators. Figure 7 depicts this scenario where 4 clients (JTAC, JFO, CAS and UAV) communicate via a publish/subscribe server located on the top node (*loganberry*) which is accessible only through the machines *lime* and *grape*. The thing to note about this scenario is that it involves contention over resources among multiple actors; user preferences are such that the machines *lime* and *grape* incur a high penalty if the load of the 4 clients is not balanced evenly between them. Each client will locally favor one of the machines based on things like the bandwidth of its links to the machines, but the clients need to coordinate to respect the even split requirement.

Table 3 shows results on this 'even split' scenario. The size of this scenario prohibits running on every DCOP representing every possible configuration of environment/system conditions, so the results shown are averages over 2000 randomly chosen configurations.

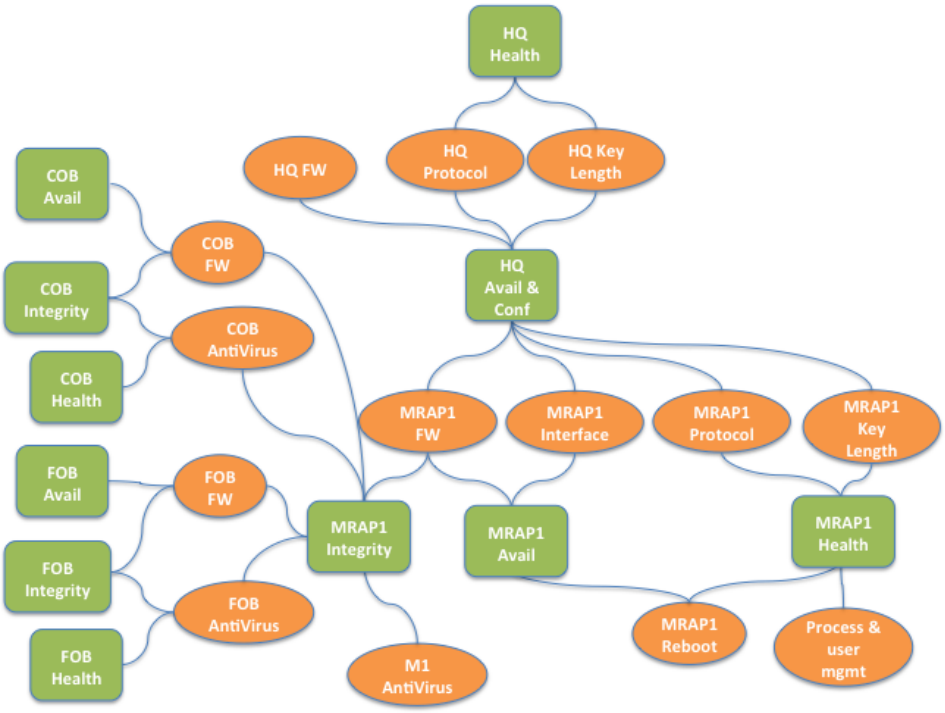

FIGURE 6. Factor graph of Scenario 6. Ovals represent variables and boxes represent functions.

TABLE 2. Time (% of F-DPOP time) and solution cost (% over optimal) on MRAP scenarios

	Scenario 1 |V|=13 |F|=7 5184 instances		Scenario 2 |V|=13 |F|=9 5184 instances		Scenario 3 |V|=8 |F|=5 20736 instances		Scenario 4 |V|=11 |F|=6 1296 instances		Scenario 5 |V|=10 |F|=5 2592 instances		Scenario 6 |V|=15 |F|=12 20736 instances	
	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost
F-DPOP	100	0	100	0	100	0	100	0	100	0	100	0
F-MS	70	13	102	13	98	15	100	15	97	29	119	10
F-ADOPT	N/A	N/A	N/A	N/A	179	0	252	0	257	0	N/A	N/A
F-DSA	88	32	133	34	194	24	139	35	143	26	139	24
F-MGM	84	29	127	29	209	17	126	30	143	19	140	21
NoMS-VP	3	29	5	46	4	16	9	30	6	16	4	45
MS-NoVP	26	3	36	2	19	3	41	0	37	2	35	2
MS-VP	28	0	37	0	23	2	43	0	40	1	37	0

As can be seen in the table, the kind of contention stemming from the even split requirement makes this set harder to solve. DSA and MGM find solutions whose cost is approximately 2.5x the optimal cost. Max-sum on its own (whether our implementation or Frodo's) results in solutions with much higher (approx. 3x) costs than DPOP's optimal. Max-sum with value propagation, however, can reach the optimal cost without significantly increasing run time compared to max-sum only, and in 40% of the time needed by DPOP. Again, ADOPT is unable to handle the larger problem size.

As we demonstrated earlier with examples, max-sum on its own can result in uncoordinated choices of values in situations with cycles and multiple local optima (in this case, any combination of 2 clients on each machine satisfies the even split requirement). Our value propagation phase was able to achieve the even split and avoid the high penalty.

To summarize, for domain-inspired instances, max-sum with value propagation gets solution quality comparable to (and sometimes the same as) DPOP's optimal quality within a fraction of the run time. And while there are cases where the benefit of the VP phase is small, it is clearly not computationally expensive.

6.2. Randomly-generated instances

Hand-generating mission scenarios (something that has to be done when real system models and

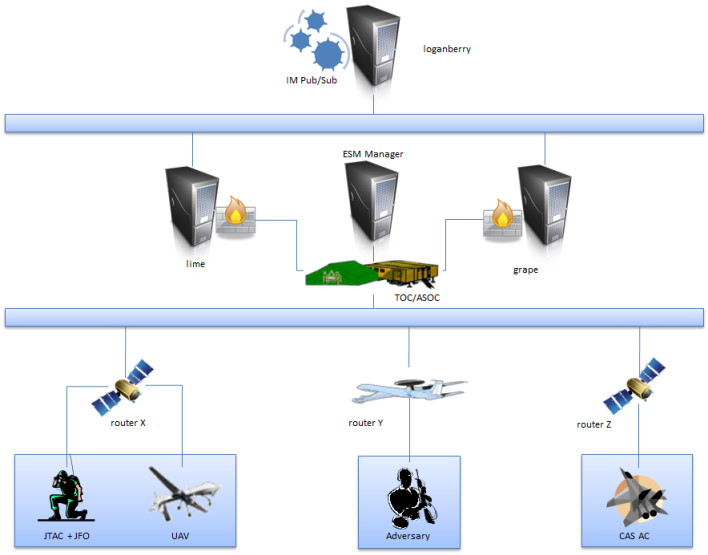

FIGURE 7. Figure 7. The 'even split' scenario

TABLE 3. Time (% of F-DPOP time) and solution cost (% over optimal) on the 'even split' scenario (2000 instances, $|V| = 24, |F| = 21$)

	Time	Cost
F-DPOP	100	0
F-MS	155	183
F-ADOPT	N/A	N/A
F-DSA	178	155
F-MGM	215	154
MS-NoVP	36	205
MS-VP	41	0

stakeholder requirements are fed into the QoS/IA runtime management framework) is a time consuming and tedious process, so the instances we manually generate are limited in size. We therefore resorted to randomly generating problem instances to better test and compare our algorithm.

Initially, we used Frodo's random generator to generate binary-constrained Max-DisCSP instances, but these lacked an important characteristic that we have in distributed QoS/IA management scenarios, which is the presence of some functions with a large scope. We therefore developed our own generator and transformed its output to XCSP format so we can still run Frodo algorithms on it for comparison. The instances we generate have loops, functions with large scopes and non-unique local minimizing assignments.

We generated 82 instances with 23 variables and 18 functions, and 82 instances with 30 variables and 25 functions. The scope of each function ranges in size from 2 to 6 and contains randomly chosen variables. Each variable has a domain of size 3 (which is the case for most variables in our domain-inspired DCOPs). To make each function have non-unique local minimizers, a function maps configurations of variables in its scope to one of only two values, with the fraction of configurations mapped to each value randomly chosen.

The results of running the various algorithms on the randomly generated instances are shown in Table 4. For DPOP, the number in brackets is the number of instances it failed to solve, and we averaged over the ones successfully solved. All other algorithms (except ADOPT) were able to solve all instances. Again, we normalize the results with respect to the time and cost of the Frodo DPOP solutions.

For the smaller of the 2 sets, local algorithms produced solutions with costs 4-5 times that of optimal, but at a third of the time needed by DPOP. The fact that MGM an DSA do not perform very well assures us that these random instance are non-trivial. Max-sum without value propagation produced solutions with very high costs. Because it is possible that the larger problem size (compared to the domain-inspired scenarios) requires more iterations for the algorithms to converge to good solutions, we ran F-MS, F-MGM and F-DSA with 160 iterations instead of 80, but results were approximately the same, so we do not report them.

For the larger of the random instances, DPOP was unable to solve 10% of the cases (we tried DPOP once with 500M and once with 1.5G of memory and it runs out of memory in both cases). On the instances that it did solve, DPOP took an order of magnitude longer time on the set with 30 variables and 25 functions than the smaller set. The larger set exacerbates the main problem with DPOP, namely the computation and space requirements a node needs to manipulate large incoming UTIL messages, which result from the large number of neighbors a variable has. Suspecting that the message size is the problem, we tried running Memory-Bounded DPOP [8], but it still could not solve the instances that DPOP failed on.

While MS-VP solution cost is 1.5-1.7x the optimal, it is 1-2 orders of magnitude faster than DPOP, which may be an acceptable quality-time tradeoff in many realistic situations. The most interesting result is that running max-sum's utility propagation phase is useless for these instances (MS-VP and NoMS-VP have the same solution quality). The cost functions are such that at the end of max-sum, the messages a variable has received do not favor any value over another. The quality of the obtained solution is solely the result of the value propagation phase which, when run without utility propagation, is one more order of magnitude faster than DPOP. We also experimented with larger instances (not shown in Table 4) with 40 variables and 160 functions. The trends were the same: DPOP was unable to solve any of the 10 instances we generated, MS-VP was 1 order of magnitude faster than MGM and 2 orders faster than Frodo's max-sum. MGM produced solutions with more than twice the cost of MS-VP.

In summary, results from the domain-inspired scenarios and the random instances show that if used after a utility propagation phase that successfully

TABLE 4. Time (% of F-DPOP time) and solution cost (% over optimal) on random instances

	|V|=23 |F|=18		|V|=30 |F|=25	
	Time	Cost	Time	Cost
F-DPOP	100	0	100 (8)	0 (8)
F-MS	101	1294	12	1485
F-ADOPT	N/A	N/A	N/A	N/A
F-DSA	38	381	5	422
F-MGM	43	437	6	460
NoMS-VP	1	53	0.1	73
MS-NoVP	8	482	1	637
MS-VP	8	53	1	73

reduces variable domains and provides each node with a good idea of the effects of its choices, our value propagation can still slightly improve solution quality without adverse effects on solution time. If performed without utility propagation, or after a utility propagation phase that results in ambiguous utilities that do not favor any subset of values, VP can still result in assignments that give solution quality comparable to that of DPOP.

7. CONCLUSION AND FUTURE WORK

Distributed systems are increasingly support missions that require higher QoS and IA requirements, two goals that are often competing for available resources. As a result of this tension, a system typically needs to consider tradeoffs between the various aspects of QoS and IA. For a distributed system, this decision making process needs to be undertaken by the nodes in the system in a distributed manner. However, the tradeoff decisions made by one node can affect QoS/IA levels delivered by another node.

In this paper, we addressed the problem of making tradeoffs in a way that maximizes the satisfaction of all users. First, we modeled the impact of the various actuator configurations available to a node on the QoS and IA attributes using cause-effect networks, then used this model to formulate the problem as a DCOP. We used the max-sum algorithm to solve our DCOPs and proposed a value propagation algorithm that helps the different nodes reach a globally consistent solution in spite of the cyclic nature of the graph and presence of non-unique local optima. Comparisons to plain max-sum, as well as other algorithms, showed that value propagation achieves solution quality comparable to optimal. More importantly, max-sum and value propagation can handle larger instances that are unsolvable using complete algorithm like DPOP and ADOPT.

The DCOP we obtain encodes the decision variables and their effects in the context of a given set of environment/system conditions. Whenever the latter change (sufficiently), the constraint functions in the DCOP are adjusted to reflect this change and the new DCOP is solved. One consequence of this approach is that from one invocation of the decision making process to the next, a stakeholder may experience a marked difference in the level of one or more QoS/IA attributes. In future work, we will try to introduce a bias towards solutions that do not cause a large change in the users' experience of the system.

Another future direction is avoiding starting the max-sum algorithm from scratch when solving the DCOP obtained from new conditions. Intuitively, some messages from the previous run of max-sum should still be useful. An existing variant of max-sum, Fast Max-Sum [17], adapts to dynamic changes in the factor graph (adding/removing a factor), but in our case, the changes are in the constraint functions, rather than the DCOP structure.

Finally, the results we obtain are only as good as the models we have. Our approach involves weights over stakeholders, preferences over attributes and a model of cause and effect. We are looking for ways to facilitate the elicitation of this knowledge from stakeholders and domain experts.

ACKNOWLEDGEMENTS

This work was supported by the United States Air Force Research Lab under contract No. FA8750-08-C-0196. Approved for Public Release; Distribution Unlimited: 88ABW-2012-4884, 10-Sep-2012. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

REFERENCES

[1] Mostafa, H and Pal, P. and Hurley, P. (2012) Message Passing for Distributed QoS-Security Tradeoffs. Proceedings of *Optimization in Multi-Agent Systems Workshop*, Valencia, Spain, June.

[2] Pal,P. and Hurley, P. (2010) Assessing and managing quality of information assurance. Proceedings of *NATO IST Symposium on Cyber Security and Information Assurance*, Antalya, Turkey, April.

[3] Hurley, P. and Pal, P. and Creti, M.T. and Fedyk, A. (2011) Continuous mission-oriented assessment (CMA) of assurance. Proceedings of *The 5th Worksop on Recent Advances in Intrusion Tolerant Systems at the 41st International Conference on Dependable Systems and Network*, Hong Kong, China, June.

[4] Farinelli, A and Rogers, A. and Petcu, A and Jennings, N. R. (2008) Decentralised coordination of low-power embedded devices using the max-sum algorithm. Proceedings of*the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 639–646, June.

[5] Bishop, C. M. (2006) *Pattern recognition and machine learning*. Springer.

[6] Petcu, A and Faltings, B. (2005) DPOP: A scalable method for multiagent constraint optimization. Proceedings of *the 19th International Joint Conference on Artificial Intelligence*, Scotland, pp. 266–271, July.

[7] Petcu, A and Faltings, B. (2004) A distributed, complete method for multi-agent constraint optimization. Proceedings of *CP 2004 Fifth International Workshop on Distributed Constraint Reasoning*, September.

[8] Petcu, A and Faltings, B. (2007) MB-DPOP: A new memory-bounded algorithm for distributed optimization. Proceedings of *20th International Joint Conference on Artificial Intelligence*, pp. 1452–1457.

[9] Farinelli, A and Rogers, A and Jennings, N. (2009) Bounded approximate decentralised coordination using the max-sum algorithm. Proceedings of *IJCAI-09 Workshop on Distributed Constraint Reasoning*, pp. 46–59, July.

[10] Zivan, R and Peled, H. (2012) Max/min-sum distributed constraint optimization through value propagation on an alternating dag. Proceedings of *The Eleventh International Conference on Autonomous Agents and Multiagent Systems*, Spain, June.

[11] Fave, F.M. and Stranders, R and Rogers, A and Jennings, N. (2011) Bounded decentralised coordination over multiple objectives. Proceedings of *The Tenth International Conference on Autonomous Agents and Multiagent Systems*, pp. 371-378, May.

[12] Xiang, Y and Chen, J. and Havens, W.S. (2005) Optimal design in collaborative design network. Proceedings of *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 241–248.

[13] Fitzpatrick, S., and Meertens, L. (2003) Distributed coordination through anarchic optimization. In Lesser, Victor; Ortiz Jr., Charles L.; Tambe, Milind (eds). *Distributed Sensor Networks: A Multiagent Perspective*, Springer.

[14] Modi, P. and Shen, W.-M. and Tambe, M. and Yokoo, M. (2005) ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161 (1-2), 149180.

[15] Maheswaran, R. T. and Pearce,J. P. and Tambe, M. (2004) Distributed algorithms for DCOP: A graphical game based approach. Proceedings of *ISCA Seventeenth International Conference on Parallel and Distributed Computing Systems*, pp. 432–439, California, USA, September.

[16] Léauté, T and Ottens, B. and Szymanek, R. (2009) FRODO 2.0: An open-source framework for distributed constraint optimization. Proceedings of *IJCAI'09 Distributed Constraint Reasoning Workshop*, pp. 160–164, California, USA, July.

[17] Ramchurn, S. D. and Farinelli, R. and Kathryn, S. and Polukarov, Maria and Jennings, N. R. (2010) Decentralised coordination in robocup rescue. *The Computer Journal*, Vol 53.

[18] Weiss, Y. and Freeman, W.T (2001) On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs. *IEEE Transactions on Information Theory*, Vol 47:2, pp. 736-744.