Message Passing for Distributed QoS-Security Tradeoffs *

Hala Mostafa
Raytheon BBN Tecchnologies
Cambridge, MA 02138
hmostafa@bbn.com

Partha Pal
Raytheon BBN Tecchnologies
Cambridge, MA 02138
ppal@bbn.com

Patrick Hurley
Air Force Research Laboratory
Rome, NY 13441
Patrick.Hurley@rl.af.mil

ABSTRACT

Information Assurance (IA) is of growing concern to the field of distributed systems. However, IA cannot be considered in isolation, as it interacts with Quality of Service (QoS); in the presence of 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 a tradeoff between IA and QoS. This tradeoff is complicated by the fact that users' relative preferences over QoS/IA aspects change based on the situation, the interests 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 tradeoff among various aspects of QoS and IA in a way that maximizes the satisfaction of all stakeholders. 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 aspects of the system in order 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 with many local configurations that 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 phase that helps nodes reach a consistent set of decisions even in cyclic factor graphs with non-unique local minimizers. We present experimental results comparing the performance of the max-sum algorithm with and without value propagation against other algorithms implemented in the Frodo [6] framework applied to two different kinds of problems.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence:**]: Distributed AI

General Terms

Algorithms, Performance, Security

*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. Approved for Public Release; Distribution Unlimited: 88ABW-2012-1017, 29 Feb 12.

Keywords

DCOP, max-sum, value propagation, QoS

1. INTRODUCTION

Multiple security mechanisms are increasingly 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. More recently, however, 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 [8].

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 QoS interfere, often adversely. For example, increasing the strength of the encryption consumes more CPU resources and increases the round trip response time for service request and response. 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 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 over IA are not static, but depend on the situation. 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 compli-

cating 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 [8, 5], 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* (CMA) approach relies on existing security and system management infrastructure to collect measurements and observations 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 there are not enough resources to meet all QoS and IA requirements of all users. We address the problem of distributed tradeoffs in this context. More specifically, we calculate tradeoffs between various aspects of QoS and IA in a way that maximizes the satisfaction of all stakeholders. For each node in the distributed system, the decision problem we consider is: what local actions to 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 as a Distributed Constraint Optimization Problem (DCOP). To do this, we first have to quantify various aspects of the system in order to be able to compare options in the course of optimization. We also need to somehow encode the effects of the various decisions on the quantities we want to optimize, a procedure that is more complex than what is generally reported in DCOP literature.

Our second contribution is modifying and extending the max-sum algorithm 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 maximize any given function, potentially resulting in locally optimal, but globally inconsistent choices. We incorporate a value propagation phase that 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 the runtime assessment and management framework that we will refer to in this paper as the QIAAMU framework (or QIAAMU for short). Section 3 briefly reviews DCOPs and the max-sum algorithm, while Section 4 details how we formulated the distributed tradeoff problem as a DCOP. We present our modifications to max-sum and our value propagation phase in Section 5. We present the results of our max-sum variants and the comparison to other DCOP algorithms in Section 6. Finally, we conclude and highlight directions for future work.

2. THE QIAAMU FRAMEWORK

The level of IA has traditionally been estimated by offline analyses, testing, modeling and experimentation. At runtime, 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).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 QoS/IA levels delivered by a system. In the QIAAMU project [8] we are developing algorithms and supporting infrastructure to enable user-specific requirement-based runtime management of QoS and IA. We have described our metrics and assessment framework in previous papers. In this paper, we focus on the decision making involved in the runtime management of QoS and IA.

Figure 1 depicts fragments of an example distributed system in which QoS and IA interfere. There are 2 MRAPs (Mine Resistant Ambush Protected vehicles) sweeping the route between a FOB (Forward Operating Base) to 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. We use this scenario to explain the following components of the QIAAMU framework:

- **Nodes:** Nodes refer to the computing and communication nodes in the distributed system. The framework uses a subset of nodes in the system as decision-making nodes/agents. The nodes in Figure 1 make up the set of decision-making nodes involved in the mission of the MRAPs, FOB and the COP. Each decision-making node is responsible for managing the IA and QoS requirements of one or more stakeholders using the actuation mechanisms that are under its local control.
- **Stakeholders:** The stakeholders are cooperating users of the system, supporting a common mission objective in different roles. For example, each MRAP has a war fighter and HQ has a commander and a system administrator.
- **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 others can be used to express 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 headquarters database, while the war fighter cares about the Timeliness of a particular service.

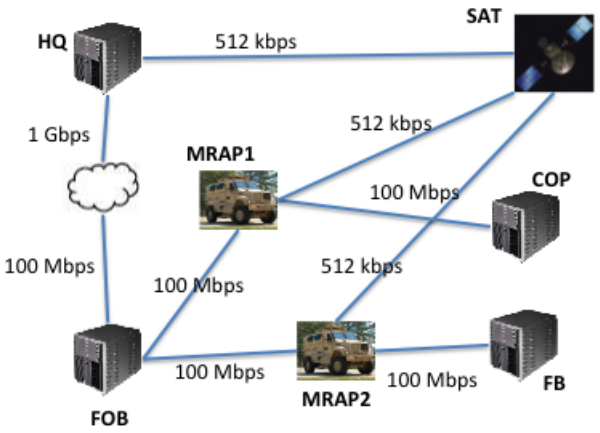

Figure 1: The MRAP scenario

- **Actuators:** The 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).

- **System/environment conditions:** The results of applying an actuator can be affected by system and environment conditions at the time of application. For example, if there is no virus threat (an environment condition), there may be no benefit in additional virus checking mechanisms in terms of whether a stakeholder's IA-related requirements are met. Similarly, if a node has been up for a long time (a system condition), rebooting the node may have a positive effect on its health. Any decisions about actuators must therefore 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?

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 $\{D_1, D_2, ..., D_n\}$. There is a set of m constraint functions (constraints) where each function 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 $\mathbf{x}$. Because we formulated our problem as a minimization problem, 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 [3]. 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

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

  where $\mathcal{M}_i$ 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

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

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

If the factor graph is a tree, max-sum is guaranteed to converge on the 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:

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

and setting itself to the value that minimizes this function.

One of the attractions of max-sum is that the size of the messages is typically small; message size scales with the size of the domains, as opposed to some algorithms where message size is exponential. 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, empirical results show that it performs well on cyclic graphs.

3.2 Other algorithms

As we show in Section 4, our DCOPs have factor graphs that contain cycles, and their constraint functions are such that variables have non-unique minimizers (the minimum of Eq 1 is attained at more than one value). The basic max-sum algorithm has no special provisions for either of these two issues, so the values chosen according to Eq 1 may give rise to an inconsistent global solution. In the following paragraphs,

we mention some algorithms or variants of max-sum that are concerned with these issues.

The issue of graphs with cycles is addressed by Distributed Pseudotree Optimization Procedure (DPOP) [10], which is a follow up for an older algorithm, DTREE [9], that is only correct for trees. In a tree, the utility reported by a node to its parent only depends on the subtree rooted at that node. In a graph with cycles, this is no longer the case. In the pseudotree created from the graph, this utility may 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, 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 pass along 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 [11].

Another approach to dealing with cycles is to remove them. Bounded max-sum [2] is an algorithm that removes cycles by eliminating dependencies between functions and variables which have the least impact on the 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 phase is somewhat similar to value propagation in Max-Sum-AD [13]. In that work, the cycles in the factor graph are addressed by assuming an order over the nodes and allowing message 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 value propagation phase. Experimental results are only reported for binary constraint functions. We plan to compare the two approaches to value propagation on problems with functions of large arity.

The work on Multi-Objective DCOPs [4] also deals with cyclic graphs with non-unique minimizers. In this case, the non-uniqueness is due to the presence of multiple objective functions, which means that there may be 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 (DPOP), or having a simple utility propagation phase but a sophisticated pre-processing of the graph into a tree (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 does not necessarily have to follow max-sum; it can be affixed to any approach that leaves each variable with a reduced domain of candidate values.

4. FORMULATING THE DISTRIBUTED TRADEOFF PROBLEM

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

4.1 Quantization and quantification

In order 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 s in the set of stakeholders $\mathcal{S}$ a weight $\boldsymbol{w_s}$ indicating the relative importance of this stakeholder. Similarly, we expresses 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-asset pair (e.g. $p_{sysadmin,Avail-HQDB}$).

Note that these values are typically going to be specific to a mission mode; a particular 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-asset pair 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-asset pair a that results from setting the actuators as dictated in the value assignment x under environment/system conditions e by $\boldsymbol{v_a^e(x)}$.

4.2 Deterministic cause-effect networks

In order 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 actuators settings and environment/system conditions, i.e., we need to elicit the various $v_a^e(x)$ functions. However, it is usually tedious and/or difficult for a user to specify each function as a flat table detailing how, for example, each configuration of firewall policies and choice of communication channel affects the Availability of HQ. It may be much easier to specify how this attribute depends on throughput and reachability and how these depend on the actuators.

We therefore use a representation of the set of functions $v_a^e(x)$ 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 represent cause-effect relations. Each non-root node has a table, which we call the Conditional Value Table (CVT), a 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 affect the Availability and Confidentiality of HQ. The agents in this example are MRAP1 and HQ. Nodes with dark borders represent environment/system conditions (e.g. the current bandwidth of the link on eth0 interface). Other top-level 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 attribute-asset pairs (e.g., Confidentiality of HQ).

This representation has the advantages of making explicit the structure of causes and effects among the various variables of interest. 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.

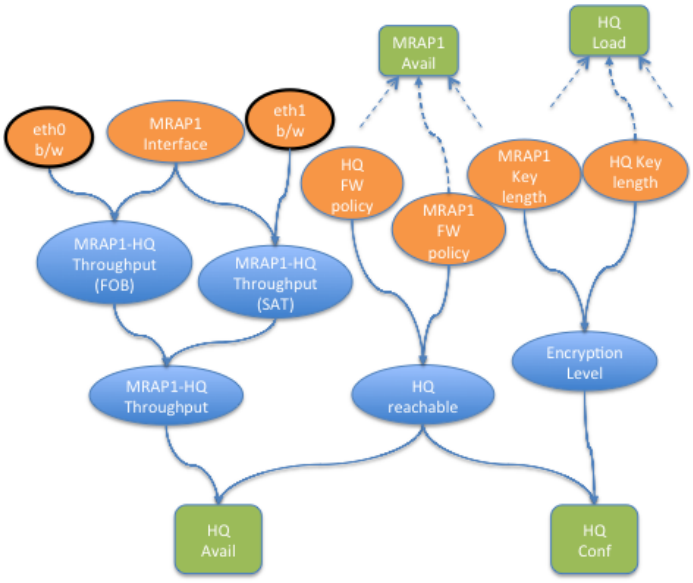

Figure 2: Part of the MRAP cause-effect network

4.3 DCOP formulation

We formulate the problem of finding the configuration of actuators that minimizes the satisfaction of all stakeholders as a DCOP. [1] 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, if there are 3 Ethernet interfaces for a node to choose from, the domain of the corresponding variable contains these 3 options.

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:

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

The second *max* operator expresses the fact that we do not care that the delivered level exceeds 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 one of which is associated with an attribute-asset pair. The cost function of pair a is therefore

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

Figure 3 shows the factor graph for the partial cause-effect network in Figure 2. As we mentioned earlier, the graphs we tend to obtain have many cycles of various length and typically have some function nodes with a large number of neighbors. In addition, the CVTs in the cause-effect network result in HQ Availability, for example, having 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 $v_a^e(x)$.

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 will typically be minimized many times under different assignments of the variables in its scope. This results in a large number of evaluations which, because of the complexity of our functions, can become an expensive process. To reduce this cost, 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 AND OTHER EXTENSIONS

Initially, we used the max-sum algorithm as described by Farinelli et. al [3] 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 space. With max-sum, we soon realized 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. The reason is that the local assignments are part of different optimal global assignments and are inconsistent with each other.

[1] 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.

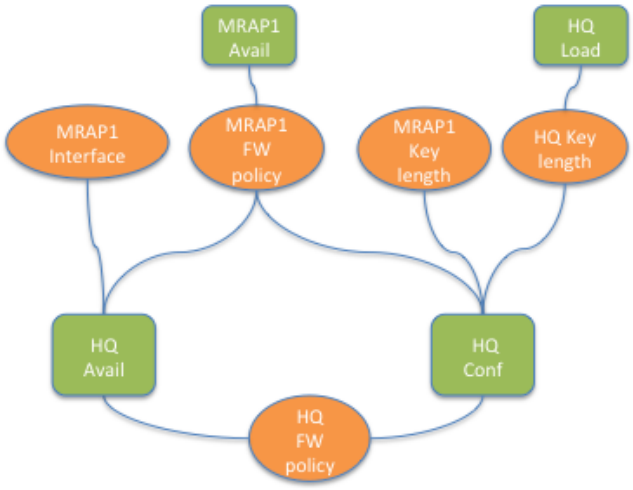

Figure 3: Part of the MRAP factor graph. Ovals represent variables (actuators) and rectangles represent functions (attributes).

An acyclic graph with non-uniqueness 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 [1]). 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 cached 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. Another problem we faced is that max-sum sometimes oscillates and fails to converge.

We experimented with adding small random noise to the values in the utility messages to break ties among the non-unique locally optimal partial configurations. We observed that even though ties were indeed 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.

To overcome the above problems, we modified the max-sum algorithm and extended it with a value propagation phase. In this phase, utilities from the max-sum utility propagation phase are used to narrow down the domains of variables. Functions then optimize in the context of these reduced domains and suggest values to their neighbors. Because the graph is cyclic, a variable can receive multiple suggestions, so it needs to decide which suggestion to adopt. This is done using a heuristic measure of function importance. If a variable ignores a function's suggestion, the later re-minimizes in order to obtain consistent values for the other variables in its scope.

The following sections detail this procedure.

5.1 Value propagation

We observed that upon the termination of the utility propagation phase, it is sometimes the case that a variable has many values minimizing the function z_i in Eq 1. However, if each variable just goes ahead and chooses one of these values, the result is a globally inconsistent solution. We therefore need a value propagation phase whose goal is to assign values in a consistent manner.

The value propagation (VP) phase starts as follows:

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*.

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 *top* function is selected (more on that later) and sent an empty VP message.

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

5.1.1 Function processing of 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 reduced, rather than the entire, domains of variables. Each reduced domain can be a single value or a set of candidates. The function would then send out VP messages to its neighbors telling them to assign themselves according to 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; we allow a variable to *renege* on a suggested value, which it does by sending a VP message back to the function with the alternative value it has taken on. The steps in processing a batch of VP messages are show in Algorithm 1.

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. They can be avoided if, during the VP phase, a function keeps 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, by setting the *sender* field to *null* (Line 4) , or 2) one of the messages conflicts with the values in *curOptimal*.

If a minimization is needed, it is followed by the function suggesting a 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) 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 is reneging on

a previous value; i.e., it already reduced its domain to one value, which is what the function used in the minimization.

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

```
1: changed = false
2: for all msg ∈ msgs do
3:    senders.add(msg.sender)
4:    if msg.sender = null then
5:       changed = true;
6:    else if curOptimal[msg.sender] ≠ msg.value then
7:       changed = true;
8:       cachedDomains[msg.sender] = msg.values
9: if ! changed then
10:   return
11: curOptimal = getMinAssignment(cachedDomains)
12: for all neighbor do
13:   if neighbor ∉ senders then
14:      neighbor.processVP(curOptimal[neighbor])
```

Another aspect of how functions process VP messages is the caching of the domains of certain neighboring variables, the reason for which will become clear after the operation of variable nodes is explained.

5.1.2 Variable processing of VP messages

Because a variable that adopts a value may later need to renege on its decision, we make each variable keep a list, $funsWhoAssigned$, of the functions that assigned it to its current value so that it can notify them if it reneges. Algorithm 2 shows how a variable processes VP messages. It starts by determining *topMsg*, the message whose sender has top priority (details below). If this sender has higher priority 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 with its value to any neighbor whom it did not receive a message from in this round (thus propagating the value to it) or who sent a message that conflicts with the new value (thus reneging on the value suggested in the message). The list $funsWhoAssigned$ is re-built in the process, in case the variable needs to renege in the future.

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

```
1: topMsg = msg whose sender is top priority
2: topFun = topMsg.sender
3: if topFun.priority ≥ maxPriority(funsWhoAssigned)
   then
4:    if currentValue ≠ topMsg.value then
5:       funsWhoAssigned.clear()
6:    currentValue = topMsg.value
7: for all neighbor do
8:    if neighbor ∈ senders and
      neighbor.msg.value = currentValue then
9:       funsWhoAssigned.add(neighbor)
10:   else
11:      neighbor.processVP(currentValue)
```

Caching domains

A function node keeps track of cached variable domains in *cachedDomains*, which is updated on Line 8 in Algorithm 1. Also, in *getMinAssignment*, if the function encounters a variable for which is it does not have a cached domain, it queries the variable for its domain, uses it for minimization, and updates *cachedDomains*. In the following, we give an example of why we need caching.

Consider a situation where f_1 is connected to v_1 and v_2 which are connected to other functions as well. Each variable has a candidate set $\{a, b, c\}$. Suppose that f_1 sent value a to v_1 and b to v_2, but that v_1 reneged on its value (because it received a suggestion from a function with higher priority) 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, since v_2 got f_1's message and adopted the value in it. What f_1 wants to use is not the new domain of v_2, but a cached version of it, if available, that potentially has a wider set of choices rather than just b. f_1's cached version of v_2's domain is $\{a, b, c\}$, which indeed allows it to explore all these options while re-minimizing.

Note that if v_2 had taken on a value suggested by a function with higher priority than f_1, it would have sent a VP messages to f_1 and in processing it, f_1 would have refreshed its cache to reflect that and used the new value in re-minimizing. So overall, caching results in the desirable behavior whereby f_1's re-minimization is done in the context of values chosen by higher priority functions, if they exist, or a broader set of options, if they do not.

5.2 Function importance and cycle detection

In the previous sub-section, we mentioned the *top* function on 2 occasions: 1) we wanted to send an empty VP message to the top function to get the VP phase started in the absence of any candidates and 2) when a variable was deciding which suggested value to take on when it receives multiple VP messages. There should therefore be a way of comparing the relative importance of function nodes.

Initially, we determined 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 assigned weights to function as follows

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

In our experiments, we only report the second approach, since it performed consistently better than the degree-based one.

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 going through the same set of states again, 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 run-time and the number of iterations. Predictably, it has no effect on solution quality. Therefore, in all the results we report for our implementation, cycle detection is on.

6. EXPERIMENTAL RESULTS

We conducted experiments to compare the following algorithms and variants:

- 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 verify that for most cases, there is value in doing max-sum.
- Max-sum without value propagation, MS-NoVP
- Max-sum with value propagation: MS-VP
- Frodo's implementation of max-sum: F-MS
- Frodo's implementation of DPOP: F-DPOP. We include this complete algorithm [10] to provide us with benchmark solution quality.

Frodo [6] is an open source framework for distributed constraint optimization that has implementations of several DCOP algorithms. For all algorithms, the times we report are running times on a single machine where agents process their messages in sequence.

6.1 MRAP scenario instances

We hand-crafted 6 cause-effect networks to depict 6 scenarios based on the MRAP mission and the underlying distributed system. Each of these DCOPs represents the trade-off 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 a few hundred, to a few thousand, DCOPs.

Figure 4 shows the factor graph of one of the larger DCOP families. Four decision makers (MRAP1, HQ, COB, FOB) decide on issues like firewall policy (FW), communication protocol, encryption key length, process and user management (single/multiple user/process allowed at a time) and ethernet interface. 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 merged 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 merged nodes 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 1: Time (in sec) and solution cost on MRAP scenarios

	Scen 1 |V|=13 |F|=7		Scen 2 |V|=13 |F|=9		Scen 3 |V|=8 |F|=5	
	Time	Cost	Time	Cost	Time	Cost
F-DPOP	174	141	176	174	131	343
F-MS	258	184	268	225	166	395
NoMS-VP	6	182	6	254	3.2	398
MS-NoVP	46	145	47	178	15.5	355
MS-VP	49	141	49	174	18.2	349

Table 2: Time (in sec) and solution cost on MRAP scenarios (cont.)

	Scen 4 |V|=11 |F|=6		Scen 5 |V|=10 |F|=5		Scen 6 |V|=15 |F|=12	
	Time	Cost	Time	Cost	Time	Cost
F-DPOP	48	23	47	42	842	124
F-MS	70	33	57	54	1194	152
NoMS-VP	2.1	30	1.7	48.6	25.3	180
MS-NoVP	9.4	23	11	42.9	204	126
MS-VP	9.8	22.7	12	42.6	212	124

Tables 1 and 2 show the times and solution costs (lower is better) produced by the various algorithms.[2] We found Frodo's timing results surprising; DPOP takes less time than max-sum and Frodo's max-sum takes much longer than our implementation of max-sum. The first observation is probably due to DPOP exchanging far fewer messages compared to max-sum, which is the strong point of DPOP. At the same time, the weakness of DPOP (the computational effort in calculating UTIL messages) is not manifested enough to make it slower than Frodo's implementation of max-sum, which takes a long time to converge. The second observation can be due to the elaborate class hierarchy that allows code reuse among the several algorithms implemented in Frodo. In fact, for the smaller networks, Frodo's DPOP took longer than a brute force centralized solver we implemented. As for the quality obtained by our max-sum implementation (without VP) compared to the Frodo implementation, it is not clear to us why ours is better.

As can be seen from the results, max-sum with value propagation gets solution quality comparable to (and sometimes the same as) DPOP's optimal quality at a small 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 one has to do when a real system model and stakeholder requirements are fed into the runtime management framework) is a time consuming and tedious process, and the instances we manually generate can only get so big. We therefore resorted to randomly generating problem instances for extensive testing

[2]Longer time is reported for a smaller scenario like Scen 3 because this scenario generated more DCOP instances.

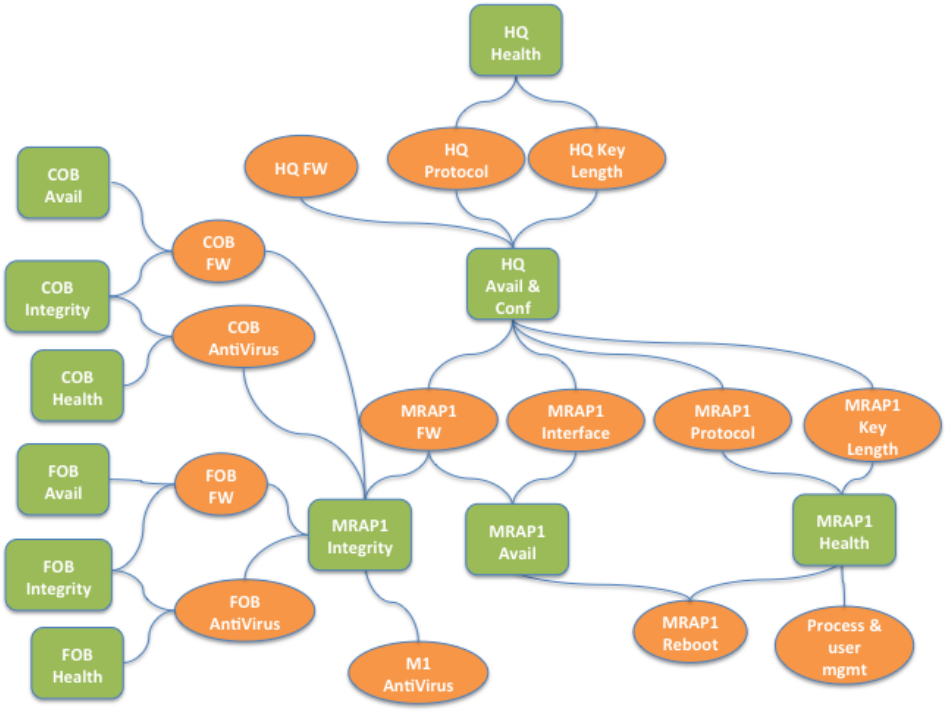

Figure 4: Factor graph of Scenario 6. Ovals represent variables and boxes represent functions.

of our algorithm.

Initially, we used Frodo's random 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 had it output the instances in XCSP format so we can still run Frodo algorithms on them. The instances we generate have loops, functions with large scopes and non-unique local minimizing assignments.

We generated 82 instances with 23 variables and 25 functions, and 82 instances with 30 variables and 25 functions. Function scopes range in size from 2 to 6. The results of running the various algorithms are shown in Table 3. We also compare to Frodo's implementation of MGM [7], a simple algorithm that should work fairly well if the instances are not too difficult, in a bid to verify that the random instances are non-trivial. For DPOP, the number in brackets is the number of instances it failed to solve. All other algorithms were able to solve all instances. The average was taken over the instances an algorithm could solve.

The larger size of these random instances compared to the MRAP scenarios instances exacerbated 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. As a result, DPOP was only able to solve a little over half the instances in the larger set. Suspecting that the message size is the problem, we tried running MB-DPOP [11], but it still could not solve the instances that DPOP failed on.

The fact that MGM does not perform very well on these random instance assures us that they are non-trivial. As with the MRAP-based instances, having a VP phase achieves quality close to DPOP's, but an order of magnitude faster. The most interesting result is the last row in the table which shows that running max-sum's utility propagation phase is completely useless for these instances. 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 3) 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.

So overall, the results from the MRAP scenarios and the random instances show that if used after a utility propagation phase that successfully 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. And if run after an unsuccessful utility propagation phase that leaves nodes unable to favor any subset of values over others, VP can still overcome this and reach solution quality comparable to that of DPOP.

7. CONCLUSION AND FUTURE WORK

Table 3: Average time (in msec) and solution cost on random instances

	|V|=23 |F|=18		|V|=30 |F|=25	
	Avg T	Avg Cost	Avg T	Avg Cost
F-DPOP	367(3)	7.4(3)	1300(37)	12.1(37)
F-MS	163	107	230	191
MGM	88	38.9	118	67.1
MS-NoVP	20	44.8	23	88.7
MS-VP	20	11.8	25	20.8
NoMS-VP	4	11.8	4	20.8

Distributed systems are increasingly catering to missions that require higher and stricter QoS and IA requirements, two forces 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 in a distributed manner by the nodes in the system. 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, then used this model to formulate the problem as a DCOP. We used the max-sum to solve our DCOPs and proposed a value propagation phase 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 maximizers. 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 an optimal algorithm like DPOP.

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.

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.

8. ACKNOWLEDGEMENT

This work has been supported by AFRL under contract No. FA8750-08-C-0196.

9. REFERENCES

[1] C. M. Bishop. *Pattern recognition and machine learning*. Springer, 1st edition, 2006.

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

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

[4] F. M. D. Fave, R. Stranders, A. Rogers, and N. Jennings. Bounded decentralised coordination over multiple objectives. In *Proceedings of The Tenth International Conference on Autonomous Agents and Multiagent Systems*, pages 371–378, May 2011.

[5] P. Hurley, P. Pal, M. T. Creti, and A. Fedyk. Continuous mission-oriented assessment (CMA) of assurance. In *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 2011.

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

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

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

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

[10] A. Petcu and B. Faltings. DPOP: A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence,*, pages 266–271, 2005.

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

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

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