Specification and Modeling of Dynamic, Distributed Real-time Systems

Lonnie R. Welch, Binoy Ravindran, Behrooz A. Shirazi and Carl Bruggeman
Computer Science and Engineering Dept.
The University of Texas at Arlington
Box 19015, Arlington, TX 76019-0015
{welch|binoy|shirazi|bruggema}@cse.uta.edu

Abstract

Time-constrained systems which operate in dynamic environments may have unknown worst-case scenarios, may have large variances in the sizes of the data and event sets that they process (and thus, have large variances in execution latencies and resource requirements), and may not be statically characterizable, even by time-invariant statistical distributions. This paper presents a specification language for describing environment-dependent features. Also presented is an abstract model that is constructed (statically) from the specifications, and is augmented (dynamically) with the state of environment-dependent features. The model is used to define techniques for QoS (quality-of-service) monitoring, QoS diagnosis, and resource allocation analysis. Experimental results show the effectiveness of the approach for specification of real-time QoS, detection and diagnosis of QoS failures, and restoration of acceptable QoS via reallocation of distributed computer and network resources.

1. Introduction

Many difficulties confront those who must engineer the emerging generation of real-time control systems. Such systems have rigorous Quality of Service (QoS) objectives. They must behave in a dependable manner, must respond to threats in a timely fashion and must provide continuous availability, even within hazardous and unknown environments. Furthermore, resources should be utilized in an efficient manner, and scalability must be provided to address the ever-increasing complexity of scenarios that confront such systems. The difficulties in engineering such systems arise from several phenomena, one of the most perplexing being the dynamic environments in which they must function. Systems which operate in dynamic environments may have unknown worst-case scenarios, may have large variances in the sizes of the data and event sets that they process (and thus, have large variances in execution latencies and resource requirements), and cannot be characterized (accurately) by constants, by intervals or even by time-invariant statistical distributions.

This paper presents techniques for specification and modeling of real-time Quality-of-Service for distributed systems that operate in dynamic environments. The techniques are based on a programming-language-independent meta-language for describing real-time QoS in terms of end-to-end paths through application programs. Constructs are provided for the specification and modeling of deterministic, stochastic, and dynamic characteristics of environment-dependent paths. The provision for description of periodic, transient and hybrid (transient-periodic) paths is also made. Another novel feature of the language is that it allows the description of multi-level timing constraints through (1) simple, cycle deadlines (which apply to a single cycle of a periodic, transient or transient-periodic path) and (2) super-period deadlines (which apply to multiple cycles of a transient-periodic path). The language and model also consider the scalability and fault tolerance features of the end-to-end paths and their application program constituents.

This is significantly different from other real-time application development languages and other real-time meta-languages. Real-time application development languages (e.g., Tomal [12], Pearl [17], Real-Time Euclid [13], RTC++ [9], Real-Time Concurrent C [7], Dicon [14], Chaos [3], Flex [16], TCEL [5], Ada95 [2], and MPL [19]) include a wide variety of features that allow assertion checking and code transformation to ensure adherence to timing constraints. Specification languages or “meta-languages” (such as ACSR [4], GCSR [1], [23], CaRT-Spec [27] and RTL [10]) formalize the expression of timing constraints and in some cases allow proofs of properties of programs based on these constraints. In [6], the features of [10] have been folded into an application development language. Our work is a specification meta-language that is application-language-independent. Rather than providing real-time support within a particular application language, it provides support for expressing timing constraints for *systems of application programs*, written in a melange of programming languages. Unlike previous work, in which timing constraints are associated at a relatively small granularity (e.g., task-level), our language allows timing constraints to be expressed the granularity of end-to-end *paths* which span many communicating programs.

Another way of categorizing real-time languages and models is in the way they permit characterization of a system's behavior when interacting with the physical environment. Prior work has typically assumed that the effects of the environment on the system can be modeled deterministically. Our work expands this to include systems that interact with environments that are either deterministic, stochastic, or dynamic. This is done by modeling interactions with the environment as data and event streams, that may have deterministic, stochastic or dynamic properties.

In order to handle dynamic environments, it is useful if the language includes features that can be related to dynamic mechanisms (such as those described in [20] [8] [21] [22] [24]) for monitoring, diagnosis and recovery. Language support for run-time monitoring of real-time programs has been addressed in [20], [6], [11], [13], and [2]. However, limited support was provided for diagnosis and recovery actions. Our work extends the language features pertaining to diagnosis of timing problems, and the migration or replication of software components to handle dynamic data stream or event stream loads (scalability). The specification language also provides support for fault-tolerance, an issue that has not been addressed in most real-time languages.

Previous real-time languages allow the description of behaviors that are purely periodic or aperiodic. Our work extends language support to describe hybrid behaviors such as the transient-periodic described in [26]. We also allow for multi-dimensional timing constraints, which, to our knowledge, cannot be described in any existing real-time language.

The remainder of the paper presents the specification language and shows how it is used for adaptive QoS management. Section 2 explains the software paradigm used in the specification language. In Section 3, we present the grammar of the specification language and illustrate its use with examples. Section 4 presents a static system model that is constructed from specifications. Section 5 defines a dynamic model, which is built at run-time by augmenting the static model with environment-dependent state. Use of the dynamic and static system models for adaptive resource and QoS management is illustrated in Section 6. Section 7 presents experimental results of applying the language, the models and the resource and QoS management techniques to a distributed real-time application.

2. The dynamic real-time path paradigm

This section presents the *dynamic real-time path paradigm*. A path-based real-time subsystem (see Figure 1) typically consists of a situation assessment path, an action initiation path and an action guidance path. The paths interact with the environment via evaluating streams of data from sensors, and by causing actuators to respond (in a timely manner) to events detected during evaluation of sensor data streams. The system operates in an environment that is either deterministic, stochastic or dynamic. A deterministic environment exhibits behavior that can be characterized by a constant value (see [19]). A stochastic environment behaves in a manner that can be characterized by a statistical distribution (see [15]). A dynamic environment (e.g., a war-fighting environment) depends on conditions which cannot be known in advance (see [22], [29]).

A (partial) air defense subsystem can be modeled using three dynamic paths: *threat detection*, *engagement*, and *missile guidance*. The *threat detection* path examines radar sensor data (radar tracks) and detects potential threats. The path consists of a radar sensor, a sensor data stream, a filtering program and an evaluation program. When a threat is detected and confirmed, the *engagement* path is activated, resulting in the firing of a missile to engage the threat. After a missile is in flight, the *missile guidance* path uses sensor data to track the threat, and issues guidance commands to the missile. The *missile guidance* path involves sensor hardware, software for filtering, software for evaluating & deciding, software for acting, and actuator hardware.

The remainder of this section describes the features of dynamic real-time *paths*. Recall that a path may be one of three types: situation assessment, action initiation, or ac-

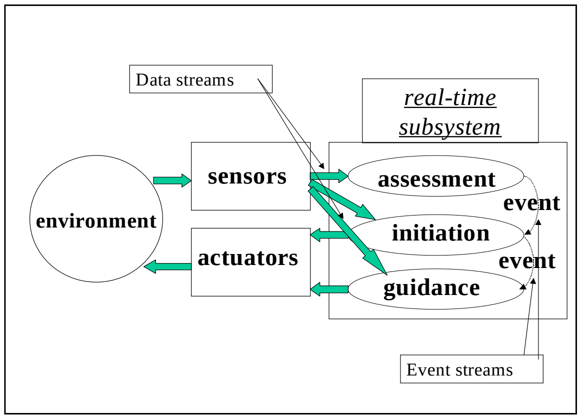

Figure 1. A real-time subsystem.

tion guidance. The first path type, situation assessment, *continuously* evaluates the elements of a sensor data stream to determine if environmental conditions are such that an action should be taken (if so, the action initiation path is notified). Thus, this type of path is called *continu-*

ous. Typically, there is a timeliness objective associated with completion of one review cycle of a continuous path, i.e., on the time to review all of the elements of one instance of a data stream. (Note: A data stream is produced by sampling the environment. One set of samples is called a data stream instance.)

The *threat detection* path of the air defense system is an example of a continuous path. It is a sensor-data-stream-driven path, with desired end-to-end cycle latencies for evaluation of radar track data. If it fails to meet the desired timeliness Quality of Service in a particular cycle, the path must continue to process track data, even though desired end-to-end latencies cannot be achieved. Peak loads cannot be known in advance for the *threat detection* path, since the maximum number of radar tracks can never be known. Furthermore, average loading of the path is a meaningless notion, since the variability in the sensor data stream size is very large (it may consist of zero tracks, or it may consist of thousands of tracks).

The second path type, action initiation, is driven by a stream of events sent by a continuous (situation assessment) path. It uses inputs from sensors to determine which actions should be taken and how the actions should be performed, notifies actuators to start performing the actions, and informs the action guidance path that an action has been initiated. We call this type of path *transient*, since it performs work in response to events. Typically, a timing objective is associated with the completion of the initiation sequence. The importance of the timing objective for a transient path is often very high, since performance of an action may be mission-critical or safety-critical.

For example, the *engagement* path of the air defense example is a transient path. It is activated by an event from the *threat detection* path, and has a QoS objective of end-to-end timeliness. The real-time QoS of this path has a higher priority than the real-time QoS of the continuous *threat detection* path.

The third path type, action guidance, is activated by an action initiation event, and is deactivated upon completion of the action. Action guidance repeatedly uses sensor inputs to monitor the progress of an actuator, to plan corrective actions needed to guide the actuator to its goal, and to issue commands to the actuator. This type of path is called *quasi-continuous,* since it behaves like a continuous path when it is active. A quasi-continuous path has *two timeliness objectives*: (1) cycle completion time: the duration of one iteration of the “monitor, plan, command” loop, and (2) action completion time (or deactivation time): the time by which the action must complete in order for success. Note that it is more critical to perform the required processing before the action completion deadline than it is to meet the completion time requirement for every cycle (although the two deadlines are certainly related). Thus, it is acceptable for the completion times of some cycles to violate the cycle deadline requirement, as long as the desired actions are successfully completed by the deactivation deadline.

The *missile guidance* path of the air defense example is a quasi-continuous path. It is activated by the *missile launch* event. Once activated, the path continuously issues guidance commands to the missile until it detonates (the deactivation event). The required completion time for one iteration is dynamically determined, based on characteristics of the threat. If multiple threat engagements are active simultaneously, the threat engagement path is responsible for issuing guidance commands to all missiles that have been launched.

3. Grammar of the specification language

This section presents a specification language for describing the characteristics and requirements of dynamic, path-based real-time systems. The language provides abstractions to describe the properties of the software, such as hierarchical structure, inter-connectivity relationships, and run-time execution constraints. It also allows description of the physical structure or composition of the hardware such as LANs, hosts, interconnecting devices or ICs (such as bridges, hubs, and routers), and their statically known properties (e.g., peak capacities). Further, the Quality-of-Service requirements on various system components can be described. In this paper, we illustrate the concrete syntax to describe only the software and the Quality-of-Service objectives.

As shown in Figure 2, a subsystem is specified by describing its priority, sets of constituent applications and devices, a set of end-to-end real-time path definitions, and a graph representing the communication connectivity of the applications and devices. The non-terminals of the subsystem grammar are explained in a depth-first manner in the remainder of this section.

As seen in the grammar of Figure 2 and the example of Figure 3, the attributes of an application include boolean variables that indicate whether the application (1) can be replicated for survivability and (2) can support scalability via load sharing among replicas. Indications of whether an application combines its input stream (which may be received from different predecessor applications and/or devices), and splits its output stream (which may be distributed to different successor applications and/or devices) are also specified. Application attributes also include all information necessary to startup and shutdown applications (not elaborated in this paper). The connectivity of the subsystem describes the flow of data and events between the applications and devices of the subsystem, and is described as a sequence of ordered application and/or device pair names within parentheses. Alternately, one may use the square bracket set notation, which is a short hand to represent a complete graph.

The definition of a path (see Figure 4) includes a set of constituent applications, various attributes, QoS requirements and data/event stream definitions. The attributes of a path include priority, type, and importance. Path type, which defines the execution behavior of the path, is either continuous, transient, or quasi-continuous. The importance attribute (a string) is interpreted as the name of a dynamically linked library procedure that may be passed arguments such as priority and the current time, and returns an integer value that represents the importance of the path.

A real-time QoS specification includes timing con-

```
<sw-sub-system>::                        Subsystem ID "{" Priority INT_LITERAL
                                         ";"  <appln-defn>+  <device-defn> { <path-
                                         defn> }* <connectivity-descr> "}"
<appln-defn>::                           Application ID "{" Scalable STRING ";"
                                         Survivable STRING ";" Combining
                                         STRING " ;"Splitting [NONE | EQUAL |
                                         STRING]";" <startup-block>+ <shutdown-
                                         block>+ "}"
<device-defn>                            Device { ID ","}* ID ";" | λ
<connectivity-descr>::        Connectivity "{" <graph-defn>+ "}"
<graph-defn>::                <pair-wise-descr> | <complete-graph-descr>
                              | ID
<pair-wise-descr>::           "(" ID "," ID ")"
<complete-graph-descr>::      "[" { ID }+ "]"
```

Figure 2. Grammar for subsystems.

```
Application FilterManager { Scalable NO;   Survivable YES;   Combining NO;
                            Splitting   EQUAL; Startup{  . . .  };  Shutdown {  . . .  } }
Device Sensor, Actuator;
Connectivity {  (Sensor, FilterManager) (FilterManager, Filter)
                (Filter, EvalDecideManager)
                (EvalDecideManager, EvalDecide) (EvalDecide, ActionManager)
                (EvalDecide,      MonitorGuideManager) (ActionManager, Action)
                (Action, Actuator) (MonitorGuideManager, MonitorGuide) }
```

Figure 3. Application, device and connectivity specifications.

straints such as simple deadlines, inter-processing times, throughputs, and super-period deadlines. A simple deadline is defined as the maximum end-to-end path latency during a cycle of a continuous or quasi-continuous path, or during an activation of a transient. Inter-processing time is defined as a maximum allowable time between processing of a particular element of a continuous or quasi-continuous path's data stream in successive cycles. The throughput requirement is defined as the minimum number of data items that the path must process during a unit period of time. A super-period deadline is defined as the maximum allowed latency for all cycles of a quasi-continuous path. A super-period deadline is specified as the name of a dynamically linked library procedure that is called dynamically to determine the estimated super-period deadline. Each timing constraint specification may also include items that relate to the dynamic monitoring of the constraint. These include minimum and maximum slack values (that must be maintained at run-time), the size of a moving window of measured samples that should be ob-

```
<path-defn>::               Path ID "{" <appn-set> <path-attribs>
                           <Real-TimeQoS><SurvivabilityQoS><stream-defns> "}"
<appn-set>::                Contains "{" { <app-name> ","}*  <app-name> "}"
<app-name>::                ID
<path-attribs>::            Priority INT_LITERAL";" Type <path-type> ";"
                                            Importance STRING";"
<path-type>::               Continuous | Transient | Quasi-Continuous
<Real-TimeQoS>::            Real-TimeQoS "{" <RTQoS-metric>+ "}"
<RTQoS-metric>::            SimpleDeadline FLOAT_LITERAL ";" <threshold>
                            | Inter-ProcessingTime FLOAT_LITERAL ";"
                                <threshold>
                            | Throughput INT_LITERAL ";"<threshold>
                            | Super-PeriodDeadline STRING ";" <threshold>
<threshold>::               MaxSlack INT_LITERAL";"
                            MinSlack INT_LITERAL ";"
                            MonitorWindowSize INT_LITERAL ";"
                            Violations INT_LITERAL ";" | λ
<SurvivabilityQoS>::        SurvivabilityQoS "{" Survivable STRING;
                                           MinCopies INT_LITERAL "}"
<stream-defns>::            <DataStream-defn> | <EventStream-defn> |
                            <DataStream-defn> <EventStream-defn>
```

Figure 4. Grammar for paths.

```
Path Sensing {
    Contains {Sensor, FilterManager, Filter, EvalDecideManager, EvalDecide}
    Type Continuous;  Priority 2;  Importance "CalcImport";
     Real-TimeQoS {SimpleDeadline  4; //secs
                    MaxSlack 80; MinSlack 20; //PERCENT
                    MonitorWindowSize 20; Violations 15;
                    Inter-ProcessingTime 7;  // secs
                    Throughput 200; // data elements per second }
     SurvivabilityQoS { Survivable YES; MinCopies 2; }
```

Figure 5. A path specification.

served, and the maximum tolerable number of violations (within the window).

Figure 4 also contains the grammar for describing survivability QoS. A survivability QoS specification includes a boolean variable that indicates (1) whether the path should be managed to ensure survivability and (2) the minimum required level of redundancy. Note that replicating a path entails replicating all of the applications that make up the path.

An example path specification is given in Figure 5. The "Sensing" path is continuous and has a priority of 2. It has a Simple Deadline of 4 seconds, which means that each review cycle must not have a latency that exceeds this amount. The MaxSlack and MinSlack definitions further constrain this requirement to the interval [0.8seconds, 3.2seconds]. If 15 out of the most recent 20 cycles violate this requirement, then corrective action by a resource manager is required. If the upper bound is exceeded, the corrective action would be to allocate more resources for the path. In the case where the lower bound is exceeded, the action would be to take away some of the resources of the path. The path inter-processing time is 7 seconds, meaning that no more than this amount of time should elapse between reviews of a particular data stream element in successive cycles. The path throughput must be at least 200 data stream elements per second. The path also requires

```
DataStream {                   DataStream {
      Type Deterministic;                        Type Stochastic;
      Size 40;}                                  Size "/home/ uta/MGDataStream.data";}

EventStream {                  EventStream {
      Type Dynamic; }                            Type Stochastic;
                                                 Rate "Exponential 0.5" ; }
```

Figure 7. Specifications of streams.

```
<DataStream-defn>:: DataStream "{" Type [Deterministic | Stochastic | Dynamic ] ";"
                               Size <environ-descr> ";" "}"
<EventStream-defn>:: EventStream "{" Type [Deterministic | Stochastic | Dynamic ] ";"
                               Rate <environ-descr> ";" "}"
<environ-descr>:: INT_LITERAL
                  | "(" INT_LITERAL "," INT_LITERAL ")" | <pdf-descr> | λ
<pdf-descr>:: STRING | FILENAME
```

Figure 6. Grammar for streams.

the existence of at least 2 copies of each of its member applications at all times.

Figure 6 shows the grammar for describing stream properties. A corresponding specification is illustrated in Figure 7. The stream type can be deterministic, stochastic, or dynamic. The data stream size or event arrival rate of a deterministic stream is a constant (scalar or interval). A stochastic stream has a data stream size or an event arrival rate that is characterized by a probability distribution function. The distribution is described as a string containing the name of a distribution and its parameters, or is described as the name of a data file containing a data set that characterizes the stream's behavior. The data stream size or event arrival rate of a dynamic stream is not described in the specification, since it must be observed at run time.

4. Static system model

This section presents an overview of a static system model (which is constructed by a specification compiler) for a single subsystem.

A software subsystem, **SS,** consists of a set of applications **SS.A** = { $a_1, a_2, \ldots$}, a set of devices (sensors and actuators) **SS.D** = { $d_1, d_2, \ldots$}, a communication graph for applications and devices **Γ(SS)** $\in \Pi((SS.D \cup SS.A) \times (SS.D \cup SS.A))$, and a set of paths **SS.P** = $\{P_1, P_2, P_3, \ldots\}$. (Note: **Π** denotes power set).

Each path **P_i** is represented as a type **$\tau(P_i)$** $\in$ {c, qc, t} (note that 'c', 'qc' and 't' denote 'continuous', 'quasi-continuous' and 'transient', respectively), a set of applications **P_i.A** = $\{a_{i,1}, a_{i,2}, \ldots\}$ (where $P_i.A \subseteq SS(P_i).A$), a set of devices **$P_i$.D** = $\{d_{i,1}, d_{i,2}, \ldots\}$ (where $P_i.D \subseteq SS(P_i).D$), a communication graph **$\gamma(P_i)$** $\in \Pi((P_i.D \cup P_i.A) \times (P_i.D \cup P_i.A))$ (note that $\gamma(P_i) \subseteq \Gamma(SS(P_i))$), a data stream **$P_i$.DS** (defined if $\tau(P_i) \in \{c, qc\}$), and an event stream **P_i.ES** (defined if $\tau(P_i) \in \{t, qc\}$). **SS(P_i)** denotes the subsystem in which path P_i is contained, **ROOT(G(P_i))** is used to denote the root application node of $\gamma(P_i)$ (the node which receives a data stream from a sensor or an event stream from an application) and **SINK(G(P_i))** is used to represent the sink application of $\gamma(P_i)$ (the application which communicates with an actuator or sends an event to the root application of another path). The type of P_i's datastream is defined as **$\tau(P_i.DS)$**$\in$ {dynamic, stochastic, deterministic}. **$\sigma(P_i.DS)$** denotes the size characteristics of P_i's datastream.

The real-time QoS requirements of a path are: required latency of **$\lambda_{REQ}(P_i)$** seconds, required throughput of **$\theta_{REQ}(P_i)$** data stream elements/time, and required data inter-processing time of **$\delta_{REQ}(P_i)$** (a maximum allowable time between processing of a particular element of P_i.DS in successive cycles). To mask momentary QoS-spikes during QoS monitoring, a specification may define a sampling window and a maximum number of QoS violations to be tolerated within the window. **$\omega(P_i)$** models the sampling window size and **$\upsilon(P_i)$** represents the maximum allowable number of violations (within sampling window $\omega(P_i)$). **$\psi(P_i) = [\psi_{min}(P_i), \psi_{max}(P_i)]$** is the required slack interval for each QoS requirement; i.e., it is required that the ratio (required QoS - actual QoS) : (required QoS) be no less than $\psi_{min}(P_i)$ and no greater than $\psi_{max}(P_i)$**.**

REPLICABLE($a_{i,j}$) --- has a true or false value to indicate if application $a_{i,j}$ can be replicated for the purpose of load sharing. **COMBINING($a_{i,j}$)** --- has the value of true or false, indicating if application $a_{i,j}$ combines the inputs from its predecessors before passing the values to successor(s). **SPLITTING($a_{i,j}$)** $\in$ (none, equal, non-equal) indicates whether an application splits it outputs among its successors, and if so, whether the outputs equally.

5. Dynamic system model

This section presents a model of such dynamic features as the current environment, its effect on QoS, a mapping of software to computation and communication hardware, and the number of replicas of each application.

The set of replicas of $a_{i,j}$ during cycle 'c' of P_i is defined as **REPLICAS ($a_{i,j}$, c) = {$a_{i,j,1}, a_{i,j,2}, \ldots$}**. The host to which $a_{i,j,k}$ is assigned during cycle 'c' of P_i is defined as **HOST ($a_{i,j,k}$, c, P_i)**, and the communication path (set of LANs and interconnection devices (ICs)) used during cycle 'c' of P_i for messages between applications $a_{i,j,x}$ and $a_{i,j,y}$ is represented as **COMMPATH ($a_{i,j,x}$, $a_{i,j,y}$, c, P_i)**.

The set of elements that constitutes a data stream can vary dynamically. **P_i.DS(c)={P_i.DS(c)$_1$, P_i.DS(c)$_2$,…}** represents the set of elements in P_i.DS during cycle 'c' of P_i. The **tactical load** (in number of data stream elements processed) of a (quasi)-continuous path P_i during it's cth cycle is **|P_i.DS(c)|**. The processing of elements of a data stream may be divided among replicas of an application to exploit concurrency as a means of decreasing execution latency of a path. In successive stages of a path that has

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming inter-face) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.