

Contribuer à Qemu

Octobre 2010

Open Wide

MODIFICATIONS

<i>VERSION</i>	<i>DATE</i>	<i>AUTHORS(S)</i>	<i>DESCRIPTION</i>
1	12/10/10	T. Monjalon	Création

Sommaire

1. Qemu pour l'embarqué.....	4
1.1. Usages.....	4
1.2. Utilisation.....	4
1.3. Debug.....	5
2. Ajouter une carte.....	7
2.1. S'appuyer sur la documentation.....	7
2.2. Comparer avec le matériel réel.....	7
2.3. Architecture.....	8
2.3.1. CPU.....	8
2.3.2. Machine.....	8
2.3.3. Périphériques.....	9
3. Implémentation.....	10
3.1. Ajout de fichiers.....	10
3.2. Variables persistantes.....	10
3.3. Fonctions basiques.....	10
3.4. Gestion d'erreur.....	11
3.5. Concurrence.....	11
4. Envoyer un patch.....	12
4.1. Conventions de codage.....	12
4.2. Mailing-list.....	12
4.3. Git.....	13
4.4. Mainteneurs.....	13
4.5. Échanges.....	14

1. Qemu pour l'embarqué

Qemu propose 3 grandes familles d'utilisation :

- émulation processeur pour une application avec intégration à l'OS ;
- émulation de périphériques pour une machine virtualisée ;
- émulation d'une machine complète.

C'est cette dernière possibilité, appelée émulation système, qui nous intéresse dans le cadre de développement de systèmes embarqués.

1.1. Usages

L'émulation d'une plateforme embarquée est un outil précieux pour de nombreux cas d'applications :

- Remplacement de matériel obsolète ou indisponible ;

Lorsqu'il n'est pas possible d'obtenir ou renouveler un élément matériel, l'émulation peut le remplacer sans modification des éléments environnants.

- Environnement de développement simulé ;

Lors des premières phases de développement d'un système embarqué, il arrive que le matériel soit indisponible car en cours de réalisation. De plus, il est beaucoup plus simple de s'abstraire des contraintes matérielles lors du développement. Cela permet un gain de temps de manipulation et une plus grande mobilité.

- Debug système non intrusif ;

À la manière d'une sonde JTAG, il est possible d'analyser l'exécution du système embarqué au moyen d'un debugger. Étant implémenté dans l'émulateur, cela ne nécessite aucune modification du système embarqué et se fait en toute transparence.

- Tests automatisés avec stimuli extérieurs ;

Le passage au "tout logiciel" ouvre des possibilités en matière de tests automatisés. En effet, Qemu permet, grâce à un système de communication bidirectionnelle (QMP), d'influer sur son comportement et de faire l'acquisition des résultats.

- Couverture de test non intrusive.

Grâce à l'outil complémentaire xcov, développé dans le cadre du projet Couverture, il est possible de tracer l'exécution du système invité et d'en déduire un rapport de couverture objet ou source. Ceci permet de répondre aux exigences de certification de manière fiable en supprimant l'étape d'instrumentation du code.

1.2. Utilisation

L'émulation complète d'une machine nécessite de compiler Qemu pour l'architecture voulue. Les architectures sont spécifiées à l'aide d'une liste. Il s'agit de l'option `--target-list` du script `configure`. La conséquence d'un ajout à cette liste est la génération d'un programme spécifique. En

effet, Qemu offre le support d'une multitude d'architecture par le biais d'autant de binaires différents. Par exemple, si l'on configure la compilation avec `--target-list='ppc-softmmu'`, on obtiendra le binaire `qemu-system-ppc`.

Chacun des programmes de la forme `qemu-system-*` peuvent émuler un certain nombre de machines. L'option `-M` permet de spécifier quelle machine doit être émulée. Pour obtenir la liste de celles qui sont implémentées, il suffit d'utiliser l'option `-M '?'`.

Chaque machine a sa propre façon de démarrer. Bien que partageant le même jeu d'options, elles peuvent nécessiter des arguments assez différents. Par exemple, certaines utilisent un BIOS livré avec Qemu, alors que d'autres nécessitent un bootloader non fourni. On peut naturellement spécifier un fichier d'image disque, un noyau ou un initrd lorsqu'ils doivent être fournis séparément. En effet, certains bootloaders peuvent accéder à un noyau inclus dans un système de fichiers alors que d'autres doivent y avoir un accès direct. C'est notamment le cas lorsque le noyau est stocké à une adresse précise de la flash.

Certaines options de la ligne de commande sont traitées de manière générique alors que d'autres sont passées en paramètre à la fonction d'initialisation de la machine qui doit les traiter de manière spécifique. C'est notamment le cas du fichier noyau, de la taille RAM ou encore du modèle de CPU.

1.3. Debug

Bien entendu, de nombreuses traces de debug sont activables dans la plupart des parties de Qemu. Il est également possible d'instrumenter le code du système invité afin d'analyser le comportement de l'émulateur.

En dehors de la bien trop célèbre méthode du `printf` et consorts, intéressons nous aux outils spécialisés : les debuggers. Dans le cas de qemu, il existe deux types de debug dynamique qui peuvent être utilisés en même temps.

Tout d'abord, comme n'importe quel programme, Qemu peut être analysé à l'aide d'un debugger du type `gdb`. On observe ainsi le comportement interne de Qemu. Cette analyse se limite souvent aux parties statiques de Qemu. En effet, un nombre important de fonctions de Qemu sont générées par le biais de jeux de macros. De plus, le moteur d'émulation processeur est un générateur de code dynamique. On voit alors rapidement les limites de `gdb`.

Mais nous avons également la possibilité de debugger certains programmes par l'intermédiaire de Qemu. En effet, un serveur pour `gdb` est implémenté et accessible par l'option `-gdb`. En utilisant le raccourci `-Ss`, Qemu attend une connexion `gdb` sur le port 1234.

Bien que le serveur `gdb` intégré à Qemu soit utilisable sur tout le système invité, l'instance `gdb` qui s'y connecte ne debug qu'un seul binaire. Ce dernier peut-être un programme en mode noyau (BIOS, bootloader, noyau) ou même le premier programme en espace utilisateur. Linux ou l'init `busybox` ont ainsi pu être analysés.

Attention, lorsque le programme traité par `gdb` n'est pas le premier à s'exécuter au démarrage, il faut utiliser un point d'arrêt dit "matériel" : `hbbreak` au lieu de `break`. Voici un morceau de script shell permettant de générer la commande `hbbreak` pour une fonction donnée :

```
ADDR=$((${PREFIX}nm $BIN | sed -n "s,^\([^ ]*\).* $FUNC$, \1,p")
```

Pour mémoire, voici la commande d'invocation de `gdb` correspondante :

```
${PREFIX}gdb $BIN --eval-command="target remote localhost:1234" \
--eval-command="hbbreak *0x$ADDR"
```

Bien entendu, les outils nm et gdb sont spécifiques à l'architecture de la cible émulée. C'est pourquoi, une variable PREFIX est utilisée.

Cette dernière technique peut être utile pour observer le comportement du système émulé et, par conséquent, analyser le bon fonctionnement de l'émulateur. Mais c'est aussi une méthode d'analyse des phases de démarrage d'un système sans avoir recours à une onéreuse sonde JTAG.

2. Ajouter une carte

Qemu sait émuler quelques cartes ARM, MIPS, PPC, etc. Elles sont, le plus souvent, des POC (preuve de concept) et permettent de tester l'implémentation du processeur qu'elles utilisent. Il y a donc un grand écart entre le nombre de cartes supportées par Qemu (une trentaine) et la quantité de matériel sortant dans le commerce chaque année. Nous sommes également très loin du nombre de plateformes supportées dans Linux.

Lorsque l'on désire émuler une plateforme avec Qemu, la première question est donc de savoir quel est le niveau de support actuel et quels seront les efforts de développement restants. Tout d'abord, il faut s'assurer que le processeur est émulé. Puis viennent les périphériques plus ou moins standards. Certains composants qui sont déjà implémentés ne sont en réalité, parfois pas testés ou encore incomplets. Mais plus souvent encore, des périphériques ne sont pas du tout supportés.

2.1. *S'appuyer sur la documentation*

L'implémentation des composants (processeur ou périphérique) manquants nécessite un travail préliminaire de documentation. C'est également nécessaire avant de compléter ou corriger un élément existant. Cette étape peut parfois s'apparenter à une "chasse" à l'information.

Il arrive que les datasheets soient fournis librement par le fabricant. Dans certains cas, il est nécessaire de signer un accord de confidentialité pour y avoir accès. Il est également possible de trouver des documents écrits à la suite de rétro-ingénierie. Si aucun document n'est disponible, il faut se référer aux implémentations des logiciels utilisant la plateforme ciblée. Le code de Linux peut être d'un grand secours dans ce domaine. Il peut être considéré comme un négatif de ce qu'il faut implémenter dans Qemu. En dernier recours, si même Linux ne supporte pas le matériel, il ne restera plus qu'à tenter une rétro-ingénierie.

Dans tous les cas, quelle que soit la source documentaire (datasheet, implémentation ou rétro-ingénierie), l'information sera orientée de sorte à permettre l'utilisation du matériel. En effet, l'exercice consistant à "écrire du matériel" de sorte à reproduire le comportement prévu par ses créateurs est inhabituel. Il n'est donc pas surprenant que les documents omettent d'expliquer le fonctionnement de certains comportements, pourvu que leur utilisation est documentée. En particulier, certaines choses sont inutiles à l'utilisation normale et ne seront donc documentées nul part. En revanche, il y a toujours un risque de trouver du logiciel s'appuyant (consciemment ou inconsciemment) sur ces parties non documentées.

Pour illustrer le cas de l'écriture d'une implémentation Qemu à partir d'un driver Linux, on peut le comparer au cas d'une application client/serveur, où l'on écrit le serveur en se basant uniquement sur le code d'un seul client.

Une implémentation matérielle se référant seulement aux matériaux documentaires ou logiciels a donc toutes les chances d'être incomplète. On parle alors de compatibilité avec des OS ; c'est-à-dire que l'émulation a été testée et ajustée de sorte à ce que certains OS puissent l'utiliser sans différence de comportement avec le matériel réel.

2.2. *Comparer avec le matériel réel*

Lorsque les documents ou OS ne suffisent pas à assurer une émulation parfaitement identique à la

réalité, il faut se référer à la source : le matériel réel.

Armé d'un oscilloscope ou d'une sonde JTAG, on peut tenter par rétro-ingénierie de comprendre son comportement.

Mais la méthode la plus confortable est sans doute de debugger le matériel à l'aide d'un logiciel qui le sollicite. Dans ce domaine, Linux est un candidat intéressant car il fonctionne sur beaucoup de plate-formes, et son code ouvert permet de le modifier pour y ajouter des remontées d'informations. Par exemple, il peut être utilisé pour se placer dans les cas limites ou non documentés.

2.3. Architecture

On peut distinguer 3 répertoires importants dans le cadre d'un portage de plateforme embarquée :

- les répertoires `target-*` pour le support des CPUs de chaque architecture, dont notamment la traduction en micro-opérations TCG ;
- le répertoire `tcg` pour la génération du code hôte ;
- le répertoire `hw` contenant la définition des machines et le support de tous les périphériques.

2.3.1. CPU

Le coeur de l'émulation est basée sur un traducteur de code machine dynamique. Le moteur d'origine s'appelait `dyngen` et est décrit dans ce document :

http://www.usenix.org/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf

Le fonctionnement est séparé en deux parties distinctes : la traduction du code invité en micro-opérations génériques puis la génération du code hôte à partir des micro-opérations. La deuxième étape est réalisée par un générateur, lui-même généré par `dyngen` à la compilation. Évidemment, les sections de codes traduits sont mises en cache pour ne pas avoir à refaire la traduction lors de l'exécution du même code.

Lors de la version 0.10 de Qemu, le système `dyngen` a été remplacé par TCG. Le fonctionnement est très similaire (en deux étapes) si ce n'est qu'il n'y a plus de générateur de générateur. Il fonctionne à présent comme un vrai compilateur dynamique et n'est plus dépendant de GCC3.

La définition des lignes d'IRQs du CPU et l'implémentation des registres spécifiques du CPU sont localisées dans le répertoire `target-*` correspondant à l'architecture du processeur. Par exemple, `target-arm` contient tout ce qui est relatif aux cœurs des processeurs ARM.

2.3.2. Machine

Les définitions de machines sont mélangées aux implémentations de périphériques dans le répertoire `hw`. Il est d'ailleurs courant que les périphériques spécifiques se retrouvent intégrés au fichier de la machine. Il n'y a pas non plus de séparation entre une machine et son microcontrôleur (ou SoC). Ce n'est pas une notion très répandue dans Qemu. Cependant, rien n'empêche d'écrire un fichier pour un SoC qui implémente les périphériques et registres internes. Ce dernier serait utilisé par une machine, au même titre que les périphériques externes au SoC (soudés sur la carte, par exemple).

Aujourd'hui, la définition d'une machine est statique : elle nécessite l'écriture d'un fichier en langage C. Ce qui signifie que si un périphérique interne est absent, c'est une nouvelle machine. Mais ce

fonctionnement devrait disparaître grâce à la définition générique des périphériques (qdev) et la configuration souple des machines nommée MachineCore. Ce sont des développements récents qui ne sont pas encore stabilisés mais qui montrent la voie qu'est en train de prendre le développement de Qemu.

Outre la gestion de ses périphériques, les machines ont la responsabilité de gérer leurs méthodes de démarrage. En effet, le support d'un bootloader ou le recours à une mémoire de masse spécifique doivent être traités comme des cas particuliers. Chaque machine possède donc une routine de chargement du noyau et de ses à-côtés.

L'allocation de l'espace de mémoire RAM est également à la charge du code spécifique de la machine. Mais cela pourrait changer prochainement.

La configuration des lignes d'interruptions est un autre sujet sensible de la définition d'une machine. Il s'agit de connecter les entrées IRQs du CPU aux sorties IRQs des périphériques. En somme, les connections faites sur le PCB ou dans le SoC doivent être retranscrites dans une table de correspondance.

2.3.3. Périphériques

La gestion des périphériques est réalisée par ce que l'on pourrait appeler des drivers. Au contraire d'un driver classique qui fait appel à des ressources telles que des registres ou de la mémoire, l'émulation de périphérique doit répondre à ces stimulations en reproduisant le comportement des ressources.

Prenons le cas d'un registre. Un driver du système invité fait un accès en lecture et l'émulation doit renvoyer la valeur du registre qui est contenue dans une variable.

Dans le cas d'un IRQ, le rôle d'un driver est de mettre à disposition une routine qui sera appelée lorsque l'interruption sera générée. Au contraire, une émulation de périphérique devra envoyer le signal d'interruption.

La nouvelle façon d'écrire un périphérique utilise un modèle objet avec héritages nommé qdev. Il permet de rationaliser l'écriture de périphérique et leur utilisation de manière dynamique (hotplug, configuration à l'exécution, etc). Markus Armbruster a fait une présentation de qdev lors du "KVM Forum 2010".

3. Implémentation

Nous allons détailler ici quelques spécificités à savoir lorsque l'on modifie Qemu.

Étant un projet portable, c'est-à-dire fonctionnant sur plusieurs OS, des couches de compatibilité sont nécessaires. De plus, un projet de machine virtuelle apporte certaines caractéristiques d'implémentation à prendre en compte.

3.1. Ajout de fichiers

Les fichiers Makefile sont construits "à la main". Les nouveaux fichiers doivent donc y être ajoutés en prenant soin de les inclure dans la bonne section du bon fichier. Ils sont découpés en domaines. En particulier, le fichier `Makefile.target` concerne les fichiers de support de machines et périphériques. Cette séparation permet de compiler les sous-systèmes en fonction de l'architecture pour laquelle ils sont destinés.

Il existe un script shell `configure` qui ne génère pas les Makefile principaux mais simplement des fichiers de configuration. Il est intéressant de noter que ce script est appelé lors d'une recompilation si les sources ont changé au point de rendre nécessaire une reconfiguration. C'est par exemple le cas lorsque l'on saute à une version différente en utilisant le gestionnaire de version (git).

3.2. Variables persistantes

La mémoire des machines et périphériques prend la forme de variables dans le code de Qemu. Celles-ci sont en général regroupées dans une structure propre à chaque composant matériel. Chaque structure représente donc, à chaque instant de l'exécution, l'état du composant. Cette organisation est à rapprocher du modèle de développement objet.

Qemu permet d'interrompre l'exécution à chaque instant et de la reprendre sans que le système invité n'en soit notifié d'aucune manière. Ceci n'est possible que si chaque élément émulé sauvegarde son état complet pour pouvoir le restaurer plus tard. Il est donc très important de mettre à disposition les fonctions de sauvegarde/restauration de la structure d'état de l'élément émulé.

3.3. Fonctions basiques

Plusieurs fonctions basiques sont proscrites dans le développement de Qemu car jugées non sûres ou remplacées par des équivalents instrumentés pour faciliter le développement. Ces contraintes sont notamment dûes au caractère portable de Qemu. Mais cela autorise également à appliquer des politiques de comportement global.

Par exemple, la fonction d'allocation mémoire `malloc` est remplacée par `qemu_malloc` qui inclut une gestion primitive d'erreur. En effet, si la mémoire est pleine, Qemu préfère s'interrompre plutôt que de chercher à résoudre le problème et dégrader l'émulation.

Ces fonctions sont décrites dans `qemu-common.h` et quelques autres fichiers associés. C'est donc une bonne idée de prendre connaissance de ses fichiers avant de commencer un développement.

3.4. Gestion d'erreur

De même que pour les fonctions d'allocation mémoire, la gestion des dysfonctionnements doit provoquer une interruption immédiate de type abort. En fonction du composant dans lequel se produit l'erreur, une fonction d'interruption spécialisée peut être utilisée afin d'afficher un ensemble d'informations permettant de comprendre le problème survenu.

3.5. Concurrency

Le fonctionnement global de Qemu repose sur une boucle principale qui fait évoluer les timers et réagit aux événements I/O. Comme l'explique Anthony Liguori dans sa présentation sur le multi-threading au "KVM Forum 2010", des problèmes architecturaux obligent Qemu à fonctionner dans un seul thread. KVM étant multi-thread, un mutex global empêche KVM de faire fonctionner plusieurs threads Qemu en parallèle. Il n'y a donc pas de gestion de concurrence à faire

Cependant, la parallélisation des tâches est en train de devenir une priorité dans Qemu. Peut-être qu'un jour, KVM pourra faire fonctionner plusieurs périphériques en parallèle. En attendant, une nouvelle notion fait son apparition : les threadlets. Ils consistent en l'utilisation de threads pour des tâches clairement identifiées comme longues ou bloquantes. Une gestion de la concurrence commence donc à faire son apparition dans ce contexte.

4. Envoyer un patch

La communauté Qemu est assez exigeante. Pour poser une question, mieux vaut envoyer un patch qui déclenchera des remarques et engagera la discussion. Mais pour qu'un patch soit intégré, il doit retenir l'attention et se conformer aux règles en vigueur.

Les méthodes sont très inspirées du projet Linux qui fait office de référence dans la construction de la communauté Qemu. En effet de nombreux contributeurs participent également à KVM, l'hyperviseur intégré à Linux.

4.1. Conventions de codage

Parfois le contenu d'un patch peut être trop spécifique pour qu'il suscite l'intérêt, donc des remarques. En revanche, la mise en forme est toujours examinée. Bien que l'ensemble du code ne soit pas complètement uniforme (en raison de son historique), la nouvelle génération de contributeurs est très attentive aux nouvelles entrées.

Afin de ne pas subir une pluie de remarques, mieux vaut donc se tenir à ce qui est écrit dans les fichiers `CODING_STYLE` et `HACKING`. Le premier ne contient que des remarques superficielles plus ou moins pénibles comme l'utilisation systématique d'accolades. On peut d'ailleurs noter que cette règle n'est pas unanimement approuvée et il est donc parfois possible d'y déroger. Le deuxième fichier concerne des méthodes d'implémentation, les fonctions proscrites et l'utilisation des types.

Bien évidemment, le bon sens passe avant toute règle édictée.

4.2. Mailing-list

La liste réservée aux développeurs (qemu-devel@nongnu.org) est un lieu d'échanges où les nouvelles fonctionnalités et les bugs sont discutés. C'est aussi l'endroit où des questions techniques peuvent être posées. Mais la plupart des discussions fertiles sont initiées par des soumissions de patches.

Tous les changements doivent être soumis par mail, sans pièce jointe, avec un titre commençant par `[PATCH]`. Lorsqu'il s'agit d'une série de patches, il est de bon ton de les numéroter et de faire en sorte qu'ils soient tous le descendant direct d'un premier mail d'introduction de la série.

Il arrive que des contributeurs ayant le droit de pousser les modifications sur le dépôt central, fassent des modifications sans passer préalablement par la mailing-list. Mais ce sont des cas isolés qui ne devraient pas être autorisés.

Si une modification implique une prise de décision ou une nouveauté architecturale, il vaut mieux l'annoncer en tant que RFC (request for comments). Il faut pour cela, insérer le sigle `[RFC]` entre crochets au début de l'en-tête du mail. Cela permet d'être un peu plus visible et d'inciter plus de gens à participer à la discussion.

Il arrive souvent que les patches doivent être retravaillés après leur première soumission. Les envois de versions successives doivent alors être numérotés afin de signaler clairement qu'il ne s'agit pas d'un nouveau patch. La forme canonique du titre ce type d'envoi est, par exemple, `[PATCH v2 0/3]` pour le premier mail de la deuxième version d'une série de trois patches.

Les relectures et suggestions sont faites dans le corps du message juste en dessous des parties de

code incriminées. Ceci est possible grâce à l'envoi de patches "inline" et non en pièce jointe.

4.3. Git

En 2009, le développement de Qemu est passé de SVN à Git. Il convient, dès lors, d'observer les règles de contribution utilisées dans un projet Git. Qemu se conforme à beaucoup de règles utilisées dans Linux. En revanche, le projet n'est pas encore aussi bien organisé que ce dernier. En particulier, il n'y a presque pas de branche de développement de sous-parties car il n'y a pas beaucoup de mainteneurs clairement identifiés pour les différentes parties de Qemu.

Il existe des branches de maintenance pour chaque version majeure, permettant d'appliquer des correctifs plusieurs mois après la livraison de la version. Mais l'ajout de nouvelles fonctionnalités se fait toujours dans la branche "master". Les patches doivent donc être "rebasés" sur la version la plus récente.

Puisque les patches doivent être envoyés "inline" et que les clients mails ont une fâcheuse tendance à casser le formatage des patches, il est recommandé d'utiliser Git pour envoyer les mails. De plus, l'utilisation de Git permet d'automatiser le formatage des mails et leur chaînage sous forme de réponse au premier mail de la série.

Pour mémoire, voici un exemple de commandes permettant d'envoyer une série de patches :

```
git format-patch --numbered --cover-letter --thread=shallow --signoff
--subject-prefix='PATCH v2' origin
git send-email --no-chain-reply-to --suppress-from --suppress-cc=author
--to=qemu-devel@nongnu.org 0*.patch
```

Le premier mail, appelé "cover letter" est pré-généré et devra être complété avant l'envoi.

Les options `--thread=shallow` et `--no-chain-reply-to` permettent de construire la série en réponse au premier mail uniquement.

Enfin, l'option `--signoff` est indispensable afin de marquer le patch comme n'étant soumis à aucune restriction autre que celle du droit d'auteur et est donc compatible avec la licence du projet.

4.4. Mainteneurs

Il est important, en arrivant dans une communauté, de comprendre quelles sont les forces en présence ; c'est-à-dire qui prend les décisions ?

Qemu a été créé par Fabrice Bellard en 2003 puis abandonné en 2008 lorsque plusieurs contributeurs ont pris la relève en tant que mainteneurs. C'est également l'époque où Qumranet (KVM) a été racheté par RedHat et a commencé la lente intégration de son fork de Qemu.

Depuis l'utilisation de Qemu par KVM en tant qu'outil de virtualisation, les entreprises dominant le secteur des serveurs sous Linux ont pris la gouvernance de Qemu. En l'occurrence, Anthony Liguori (IBM) est le nouveau leader de Qemu, tandis qu'Avi Kivity (Qumranet/Red Hat) est le leader de KVM.

La plupart des contributions s'orientent donc vers la capacité de virtualisation. La capacité de Qemu à être utilisé dans l'embarqué, grâce à TCG, n'a pas une place aussi importante. Pourtant Anthony Liguori continue de défendre le maintien de TCG et tous les usages qui en sont fait.

4.5. Échanges

Bien que l'essentiel des discussions ont lieu sur la mailing-list, il existe de nombreux autres lieux de communications pour les contributeurs de Qemu. Certaines conversations, souvent moins importantes, prennent place sur le salon IRC. Les décisions importantes sont discutées lors d'une réunion téléphonique qui a lieu presque toutes les semaines. La préparation et le compte-rendu de ces réunions se retrouvent souvent sur la mailing-list.

Bien que peu utilisé, le wiki permet également de communiquer. C'est en particulier le lieu idéal pour présenter les projets en cours ou terminés. Il tient également lieu de portail d'accueil.

On peut enfin citer les rencontres qui ont lieux lors de conférences. Lors de l'été 2010, un Forum KVM a réuni une grande partie des contributeurs régulier de Qemu. C'était l'occasion de présenter les évolutions en cours ou à venir. De plus, une conférence aura lieu en France au début de l'année 2011 : le Qemu Users Forum.